

Parsing Transformative LR(1) Languages

Blake Hegerle

February 7, 2008

Abstract

We consider, as a means of making programming languages more flexible and powerful, a parsing algorithm in which the parser may freely modify the grammar while parsing. We are particularly interested in a modification of the canonical LR(1) parsing algorithm in which, after the reduction of certain productions, we examine the source sentence seen so far to determine the grammar to use to continue parsing. A naive modification of the canonical LR(1) parsing algorithm along these lines cannot be guaranteed to halt; as a result, we develop a test which examines the grammar as it changes, stopping the parse if the grammar changes in a way that would invalidate earlier assumptions made by the parser. With this test in hand, we can develop our parsing algorithm and prove that it is correct. That being done, we turn to earlier, related work; the idea of programming languages which can be extended to include new syntactic constructs has existed almost as long as the idea of high-level programming languages. Early efforts to construct such a programming language were hampered by an immature theory of formal languages. More recent efforts to construct transformative languages relied either on an inefficient chain of source-to-source translators; or they have a defect, present in our naive parsing algorithm, in that they cannot be known to halt. The present algorithm does not have these undesirable properties, and as such, it should prove a useful foundation for a new kind of programming language.

1 Introduction

Programming is the enterprise of fitting the infinitely subtle subjects of algorithms and interfaces into the rigid confines of a formal language defined by a few unyielding rules—is it any wonder that this process can be so difficult? The first step in this process is the selection of the language. As we go along in this enterprise, we might find that our selected language is inadequate for the task at hand; at which point we can: forge ahead with an imperfect language, we can attempt to address the problematic section in a different language, or we can jettison the language for another with its own limitations, thereby duplicating the effort already put into writing the program in the first language. With ever larger, more complex programs, we increasingly find that no single language is especially well-suited—yet if we try to use multiple languages, we face significant hurdles in integrating the languages, with rare exceptions. A fourth possibility presents itself: we could create a new programming language that contains all of the features we will ever need

in any section of the program; aside from the fact that creating a general programming language is a monumental effort in and of itself, the resulting programming language will likely be a cumbersome monster. What we seek is a language that is at once general enough to suffice for very large programs, while also having specific features for each portion of the program.

There are a great deal of mature programming languages in existence, each with its own advantages and disadvantages. None of these are the language we seek. Ideally, we would like to be able to take an existing programming language and—without having to duplicate the tremendous amount of effort which went into its creation and development, not to mention our own effort in learning it—mold it to our needs.

We do not have time to survey the major languages, but programs in these languages do fit a general mold: programs must be syntactically well-formed, then they must be semantically well-meaning, and finally, they must specify a program that is free from run-time errors. Moving from the source code for a program to a run-time executable involves three phases: syntax analysis, semantic analysis, and code generation. The first two analysis phases are not separated in practice, but are performed in concert by a parser which is generated by a parser generator. The parser generator takes a grammar describing the syntax of the programming language, in addition to the semantic value of each production in the grammar, from which it produces a parser. If we had the source code to the compiler, we could change it to suit our purposes, producing a **derived language**. However, we must be careful if we do this, for changes to the code generator could produce binaries that lack compatibility with existing binaries.

The aforementioned approach is not terribly common: its most glaring problem is that a program written in a derived language cannot be compiled by a “normal” compiler. An alternative is to make a new compiler wholesale—one that, rather than outputting a binary, outputs source code in an existing programming language; such a compiler is known as a **source-to-source translator**. The practice of creating source-to-source translators is much more common than the practice of creating derived languages; two examples are the cfront compiler for C++ and a WSDL compiler for SOAP. These two examples illustrate an interesting point: the new language can share much with the target language, as is the case with cfront; or, the new language can share nothing with the target language, as is the case with a WSDL compiler.

Creating a derived language is an attractive concept because we can directly leverage an existing implementation of a base language, but modifying any large program—the compiler, in this case—in an ad-hoc manner is not exactly an easy task. This approach becomes decidedly less attractive if we seek to radically alter the language: we will likely find that the code generator is tightly coupled with the parser, and that the facilities for creating abstract syntax trees have no more generality than is necessary for the original language. Creating a source-to-source translator, on the other hand, is an attractive concept because we can make a language that departs from the target language as much or as little as we want; however, perhaps too much information is lost in the conversion to the target language: data such as debugging information, higher-order typing, optimization hints, and details necessary for proper error handling are just a few of the things which might get lost. Another problem is what I refer to as the “language tower problem”: say we start with a language L, then we create a source-to-source translator from L++ to L, then we create a source-to-source translator for Aspect-L++, then we create a

source-to-source translator for Visual Aspect-L++, ad nauseam—in short, we end up with far too many parsers.

We will take an approach somewhere between these two. We would like to develop a **base language** that is general purpose enough to serve us in its unmodified form, yet can be modified at our pleasure. In light of our consideration of derived languages, we will create a general-purpose framework for abstract syntax trees that both the base language and any derived language can use to capture the full range of the semantics of a program. Also, we will not require someone wishing to create a derived language to create an entire grammar: we will allow modifications of the existing grammar. We have avoided most of the problems associated with source-to-source translation as well: the data. As an example consider: run-time metadata (like profiling and/or debugging data), data type, optimization hints, and error messages; none of these are likely to be present in the binary if source-to-source translators are used. Since we are making it easy to modify the language, we would expect that the language tower problem would be exacerbated, but this is hardly the case, for there is only ever a singular parser.

We pause to note that there must be some way of specifying the semantic actions of a production. We can assume that these actions are specified in a programming language, probably the base language itself, and that the parser has an interpreter for that language included in its implementation.

The code generator only understands so much of what is potentially in an abstract syntax tree. Everything else—the debugging, type, optimization, and error data—which gets added to the tree must be, to a large extent, ignored by the code generator. However, these data—we will call them **extended semantics data**—are not valueless; thus, we will allow additional analysis phases to be performed on the abstract syntax tree between that parsing and the code generation phases. Here again, an interpreter embedded in the parser will be invaluable.

How might a language like this be used? We can use it to add gross language features, for example object-oriented or aspect-oriented support. Or we could add more behind-the-scenes features, improving for example, the optimizer. If we know that we are using a particular library, we can give first-class syntactic support to common patterns—for example, we could support monitors, as Java does with the `synchronized` keyword. Finally, we could create a modified grammar to eliminate repetitive code, using the modifiability of the language as a sort of macro processor.

We will allow the parser to modify itself *during parsing*. From here on, we will assume that a parser operates strictly left-to-right. No longer can we treat the syntax analysis and semantic analysis phases as entirely separate, even conceptually, for some part of a file may define the syntax and semantics of the remainder of the file.

The study of formal languages has produced many interesting classes of languages: regular, context-free, context-sensitive, and recursive-enumerable being the best known. If \mathcal{X} is a class of languages, then the set of **transformative** \mathcal{X} languages are those languages whose strings x can be decomposed as $x = y_1 y_2 \dots y_n$, such that y_i is a substring of an element of one of the languages in \mathcal{X} , which we term the i^{th} **instantaneous language**; further, y_i specifies the instantaneous language $i + 1$.

Our goal in the present work will be to develop a method of parsing a useful class of transformative languages. Our parser will operate much like a classical parser, except that, as it moves over the boundary between y_i and y_{i+1} , it will mod-

ify itself—more precisely, it will modify its grammar, and then its parsing tables. Since we are dealing with a self-modifying parser, we would run into problems if the parser were to backtrack from y_{i+1} to y_i —not insurmountable problems, to be sure, but we will find a satisfactory non-backtracking method of parsing that does not have these problems.

1.1 Applications of Transformative Parsing

Let us say that we need to write a graphical program in a language much like Java which can access both a web service and a database; let us assume that we do not have any visual rapid-development tools. We must write a lot of GUI code like “make a window, put a layout in the window, put the following controls in the layout: . . . , add a toolbar to the window, add an item to the toolbar with the label ‘x’, set the callback object to ‘y’” etc. We must write a lot of database like “parse a query, bind the following variables (. . .), execute the query, create a cursor, advance the cursor, get the first column, get the second column” etc. We must write a lot of web service code like “create a procedure call, marshal the input, call the procedure, demarshal the output, handle any exceptions” etc. The GUI, database, and web service functionality is most likely handled by a library. Would that each library added syntax constructs for the operations it provides. We could declare the GUI with code like

```
window{ layout{...}; toolbar{ item{ label = 'x'; action = y } } }
```

The interesting thing is that the `y` identifier is bound to the correct lexical scope.

We could process our query with code like

```
query(select col1, col2 from t1 where col=$z -> (c1, String c2) {  
    ...  
}
```

Here, `z` is a variable in the scope containing the `query` construct, as is `c1`; however, `c2` is local to the block after the `query`.

Finally, we could process our web service with code like:

```
webservice service=ws_connect{ url=http://www/shop, id=shop };  
a=shop.lookup(b, c);
```

There is nothing to prevent the compiler from doing a compile-time type-safety check: is the column `col` on the table `t1` the same type as the variable `z`, or is the column `col1` the same type as `c1`? Something similar can be done for the `webservice`. Is the second argument of the `lookup` method of the `webservice` the same type as `c`?

The way that we arrive at the functionality requirements for these examples is to allow the library writer to specify new syntactic constructs to simplify complex, error-prone tasks—along with this syntax, there must be provided a semantic description of the new construct. A perfectly viable way to specify semantics is as YACC and ANTLR do: associating a block of procedural code with each production, which will be run after that production is used.

We conclude with the observation that the capabilities of the hypothetical system under discussion in this section are not really new. Indeed, for the web service example at least, the capabilities are common. However, continuing with that example, the method used to achieve web service integration with the host language is by means of a WSDL compiler; one of our goals is to make obsolete artifacts

like WSDL compilers. What *is* new is a single mechanism which integrates the language with the program.

1.2 Conventions

Will will also observe some (fairly standard) typographical conventions to denote the type of variables. See Table 1. We often deal with grammars in the sequel, which have either 4 or 6 components; in most cases, we will label the components of these grammars as (Σ, N, P, S) and (Σ, N, P, S, T, M) , as appropriate.

1.3 The Rest of This Document

There are 4 main sections in the sequel. In Section 2, we present the formal definition of a transformative language—this definition is perhaps surprisingly complex, but it allows us to establish Theorem 2, a result which states that parsing sentences in an appropriate transformative language can be done in a finite amount of time.

The Algorithm to recognize sentences generated by a given grammar we present and justify in Section 3. We rely on the determination that a particular object is in a particular set—the object being a transformation (i.e., a change of grammar) being what we term “valid.” In Section 4, we present an Algorithm to test a transformation for membership in the aforementioned set of valid transformations. We spend a great deal of time proving that this Algorithm is correct.

Finally, in Section 5, we survey related work: there are the previously mentioned “extensible languages;” other investigations into parsers whose grammar changes while parsing, frameworks for creating derived (from Java, usually) languages, macro systems are related in this Section. There are also a few works very near to the present one.

2 Transformative LR(1) Languages

The $LR(k)$ class of languages, where $k \geq 0$, is the largest known class of languages which can be parsed without backtracking. The theory of these languages, along

terminals	a, b, c, d, e, f
nonterminals	A, B, C, D, E, F
terminal strings	$r, s, t, u, v, w, x, y, z$
symbol strings	$\alpha, \beta, \gamma, \delta, \zeta, \eta, \theta, \kappa, \lambda$
symbol	U, V, W, X, Y, Z
start symbol	S, S'
production	π, τ, ϕ
node	A, B, C, D, E, F
set of nodes	U, V, W, X, Y, Z
transformation	Δ
set of transformations	$\mathcal{V}, \mathcal{T}, \mathcal{D}$
a grammar	G

Table 1: Typographical conventions.

with an algorithm to determine that a grammar is $LR(k)$, are due to Knuth [21], but we will primarily draw upon the presentation of this theory from [2]. We will not concern ourselves with the more general case of $LR(k)$ languages for $k \neq 1$. It shall turn out to be the case that $LR(1)$ languages are convenient as a basis for creating a practical transformative language.

We first review $LR(1)$ parsing. We then try to extending the $LR(1)$ parsing algorithm in the most naive way possible. The first attempt we make will not be successful, but it will illustrate a subtle problem in the development of transformative $LR(1)$ languages. Rectifying these problems will occupy us for much of the rest of this work.

2.1 LR(1) Languages

The $LR(1)$ class of languages are a subset of the context-free languages. Context-free languages are those that are generated by a context-free grammar, which is a tuple (Σ, N, P, S) , where Σ is the terminal alphabet, N is the nonterminal alphabet, P is the set of productions, and S is the start symbol. We establish the convention that there is a special symbol $\dashv \in \Sigma$, that does not appear on the right side of any production; this symbol is used to terminate strings in the language. We will specify $LR(1)$ languages by presenting a context-free grammar for that language; given a context-free grammar, we cannot tell at first glance whether or not the grammar specifies a $LR(1)$ grammar, rather, we must utilize the tools of $LR(1)$ theory to make this determination.

We must clarify some notation we will have occasion to use. Let $G = (\Sigma, N, P, S)$ be a context-free grammar. Since “ \rightarrow ” is a binary relation, it is in fact a subset of the cartesian product $N \times (\Sigma \cup N)^*$; this subset is P . If $A \rightarrow \alpha$ is a production, then there is an ordered pair $(A, \alpha) \in P$. We have no problem with notation like: $\pi = (A, \alpha)$; we therefore ought to have no problem with notation like: $\pi = A \rightarrow \alpha$. We take the symbol \xrightarrow{G}^* to mean the replacement of the rightmost nonterminal in a string, and we take \xrightarrow{G}^* to be a rightmost derivation of zero or more steps.

One way to define $LR(1)$ languages is via Definition 3, which, along with Definition 1, is from Chapter 5 of [2].

Definition 1. If $G = (\Sigma, N, P, S)$ is a context-free grammar, then for any $\alpha \in (\Sigma \cup N)^*$, we define $\text{FIRST}_k(\alpha)$ to be the set of all $y \in \Sigma^*$, where $|y| = k$, such that $\alpha \xrightarrow{G}^* yx$ for some $x \in \Sigma^*$. We understand $\text{FIRST}(\alpha)$ to mean $\text{FIRST}_1(\alpha)$. We call $\text{FIRST}(\alpha)$ the **first set** of α .

Definition 2. Let $G = (\Sigma, N, P, S)$ be a context-free grammar, and let S' be a nonterminal not in N . Define the context-free grammar $(\Sigma, N \cup \{S'\}, P \cup \{S' \rightarrow S\}, S')$ as the **augmented grammar** associated with G .

We are interested in augmented grammars because it is an easy way of ensuring that the start symbol does not appear on the right side of any production: this is a necessary condition for the construction of $LR(k)$ parser tables.

Definition 3. Let $G = (N, \Sigma, P, S)$ be a CFG and let $G' = (N', \Sigma, P', S')$ be its augmented grammar. We say that G is $LR(k)$, $k \geq 0$ if the three conditions

1. $S' \xrightarrow{G'}^* \alpha Aw \xrightarrow{G'}^* \alpha \beta w$,
2. $S' \xrightarrow{G'}^* \gamma Bx \xrightarrow{G'}^* \alpha \beta y$, and

3. $\text{FIRST}_k(w) = \text{FIRST}_k(y)$
 imply that $\alpha Ay = \gamma Bx$. (That is, $\alpha = \gamma$, $A = B$, and $x = y$.)

2.1.1 Shift-Reduce Parsing

A shift-reduce parser is a deterministic pushdown automaton with a stack of binary tuples, controlled by a **parsing table**, calculated from the language's context-free grammar before parsing begins. Each tuple on the stack is in the set $\mathbb{Z}_k \times (\Sigma \cup N \cup \{\epsilon\})$, where K is a finite set of integers. The parser can do one of three things:

1. it can remove the first symbol from the input string, and put it and a state onto the stack, a move which we will call a **shift**;
2. it can remove zero or more tuples from the stack, and replace them with a single new tuple, a move which will will call a **reduction**; or,
3. it can halt.

Should the automaton be in an accepting state when it halts, then we know that $x \in L(G)$, and we say that the parser **accepts** the string x . Should the parser halt in any other state, then we know that $x \notin L(G)$, and we say that the parser **rejects** the string x . The parser is in an accepting state if and only if it just reduced by the production $S' \rightarrow S$.

The automaton examines the current input symbol, which will will call a , and takes an action based upon the value of a and the value of the integer in the tuple on top of the stack, which we will call k ; if $\text{action}[k, a] = \text{shift } m$, then we set a to be the next input symbol and we push (m, a) onto the stack; if $\text{action}[k, a] = \text{reduce } "A \rightarrow \alpha"$, then we pop $|\alpha|$ states off of the stack and, letting (k', X) be the tuple on the top of the stack after popping those items off, we push $(\text{goto}[k', A], A)$ onto the stack; if $\text{action}[k, a] = \text{error}$, then we halt in a non-accepting state; finally, if $\text{action}[k, a] = \text{accept}$, then we halt in an accepting state.

2.1.2 The Canonical Shift-Reduce Parser for an LR(1) Grammar

As we just saw in Section 2.1.1, all of the decisions on how to parse a string are deferred to the construction of the parsing tables. It is this construction we turn to now.

Given a context-free grammar, there are many ways of constructing parsing tables for a shift-reduce parser. Not every method will succeed for a given context-free grammar, but if a grammar is LR(1), there is one method which is guaranteed to work: this is the original method of Knuth [21], and we refer to the parser (the tables used by the shift-reduce parser, specifically) produced by this method as the **canonical LR(1) parser** for that grammar.

The previously cited source does present the algorithm for the construction of LR(1) parsing tables [21], but in an indirect form; a more direct presentation is [2]. Perhaps the most friendly presentation of this algorithm is in [1, chap. 4]. We need only summarize the algorithm here, following the presentation from [36]. Let $G = (\Sigma, N, P, S)$ be a context-free grammar. We begin by augmenting the grammar. We then construct sets of **LR(1) items**; in general, these items are of the form $[A \rightarrow \alpha \cdot \beta, a]$, where $A \rightarrow \alpha\beta$ is a production, and $a \in \Sigma$. Intuitively, we think of an item as a memo to ourselves that we are trying to match the production $A \rightarrow \alpha\beta$, we have so far matched the α , and we expect to match β later; the meaning of the

“ a' ” is this: when we have matched $\alpha\beta$, we reduce by $A \rightarrow \alpha\beta$ if and only if the lookahead is a . We begin our construction of the item sets with the **initial item** $[S' \rightarrow \cdot S, \cdot]$; we let I_0 be the closed item set containing the initial item, where we define an item set to be **closed** if for every item of the form $[A \rightarrow \alpha \cdot B\beta, a]$ in that set, such that $B \rightarrow \gamma$ is a production, we have that $[B \rightarrow \cdot \gamma, b]$ is in that item set, for all b in $\text{FIRST}(\beta a)$ (see Definition 1). For item sets I_k and I_m , and a grammar symbol $X \in (\Sigma \cup N)$, we define the **goto function** on I_k and X to be the closed item set I_m , which we write as $m = \text{goto}[k, X]$, if $[A \rightarrow \alpha \cdot X\beta, a] \in I_k$ and $[A \rightarrow \alpha X \cdot \beta, a] \in I_m$. Finally, we define the **action function** for an item set I_k and a terminal a in one of two ways: if there is an item $[A \rightarrow \alpha \cdot a\beta, b] \in I_k$, then we let $\text{action}[k, a] = \text{shift } m$, where $m = \text{goto}[k, a]$; otherwise, if there is an item $[A \rightarrow \alpha, a] \in I_k$, then we let $\text{action}[k, a] = \text{reduce } "A \rightarrow \alpha"$. The only exception to this last rule is if the item is the initial item: if we reduce by the production $S' \rightarrow S$, then we **accept** the string, or recognize that the string is a member of the language.

We can now encode the functions `goto` and `action` into two tables, as suggested by the bracketed notation. For any entry on the action table corresponding to an undefined value of the action function, we give that entry the value of “error”. These are the **canonical LR(1) parsing tables**.

2.2 A Note on Parse Trees

If we have a context-free grammar $G = (\Sigma, N, P, S)$ and we have some $x \in \Sigma^*$, then we can prove that $x \in L(G)$ by supplying a derivation of the form

$$S \implies \alpha_1 \implies \alpha_2 \implies \dots \implies \alpha_n = x. \quad (1)$$

Definition 4. If T is a tree such that:

1. the root of T is labeled S ;
2. every interior node, with label X , the children of which are labeled Y_1, Y_2, \dots, Y_n , such that $X \rightarrow Y_1 Y_2 \dots Y_n$ is a production;
3. every leaf is labeled with a terminal or ϵ ; and
4. the yield of T (removing ϵ 's) is x ,

then we say that T is a **simple parse tree** for x .

The existence of a simple parse tree for x is a necessary and sufficient condition for $x \in L(G)$.

Definition 5. Let T be a tree whose nodes are labeled with terminals, productions or ϵ . Let $\mathcal{R}(T)$ be the root of T . For any node N , define

$$\mathcal{L}(N) = \begin{cases} a & \text{if } N \text{ is labeled with terminal } a \\ \pi & \text{if } N \text{ is labeled with production } \pi, \\ \epsilon & \text{if } N \text{ is labeled with } \epsilon \end{cases}$$

and define

$$\mathcal{L}_H(N) = \begin{cases} a & \text{if } N \text{ is labeled with terminal } a \\ A & \text{if } N \text{ is labeled with production } A \rightarrow \alpha. \\ \epsilon & \text{if } N \text{ is labeled with } \epsilon \end{cases}$$

If N is an interior node labeled with the production $B \rightarrow \alpha$, whose children are M_1, M_2, \dots, M_m , then we define two more functions— \mathcal{C} and \mathcal{S} —as follows: define

$$\mathcal{C}(N) = \mathcal{L}_H(M_1)\mathcal{L}_H(M_2)\dots\mathcal{L}_H(M_m),$$

and define

$$\mathcal{S}(N) = \alpha.$$

We can reformulate Definition 4 as follows.

Definition 6. If T is a tree such that:

1. every interior node is labeled with a production;
2. if N is an interior node, then $\mathcal{C}(N) = \mathcal{S}(N)$;
3. every leaf is labeled with a terminal or ϵ ; and
4. the yield of T (removing ϵ 's) is x ,

then we say that T is a **parse tree** for x .

We developed the nonstandard definition of a parse tree in Definition 6 because we will have occasion, particularly in Section 4, to do nontrivial work on parse trees that would be impossible with simple parse trees, as removing child nodes from a node destroys information about that node.

Definition 7. Let T be a tree and let N and M be nodes in T , such that N is an ancestor of M . One of the children, which we will call A , is a child of N such that either A is M itself, or A is an ancestor of M . In either case, we say that A is **autoancestral** to M .

We finish with a note on ordering parse trees. A parse tree is inherently an ordered tree. If nodes N_1 and N_2 share the same parent, then N_1 and N_2 are comparable; call this partial order \leq_T . We will find it convenient to give parse trees the following total order.

Definition 8. Let an ordered tree be given, with A and B nodes in that tree. Define $A \leq B$ if any of the following are true:

1. $A = B$;
2. B is a descendant of A ; or
3. there exist distinct nodes C and D with a shared parent and $C \leq_T D$, such that C is autoancestral to A , and D is autoancestral to B .

Our total ordering of the nodes in a tree suggests a method of diagramming trees. We can illustrate this, along with some of the other ideas of this section, with an example. Let $G = (\{a, b\}, \{S\}, P, S)$ be a context-free grammar, with $P = \{S \rightarrow aSb \mid \epsilon\}$. Consider the derivation of the string $aabb$:

$$S \implies aSb \implies aaSbb \implies aabb.$$

We can represent this using the diagram in Figure 1; in that diagram, nodes appear least to greatest from top to bottom.

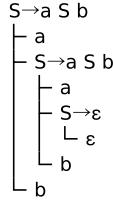


Figure 1: The parse tree for the string *aabb*.

2.3 Grammars and Transformations

When the grammar is changed during a parse, we will often want to change only a part of the grammar, rather than the entire grammar. Therefore, we will treat a change of grammar as the act of adding and removing productions from the grammar; we call this act a **transformation of grammar**. Only after reduction by certain productions will we add or remove productions from the grammar; we call these productions **transformative productions**.

We can begin to see how the parsing algorithm will have to be modified to parse a transformative language: when the parser performs a reduction, it checks to see if the production was a transformative production; assuming that it was, we perform a transformation of the grammar and calculate the parsing tables for the new grammar. The parsing stack is completely dependent upon the particulars of the parsing tables; hence, when we stop to perform a transformation of grammar, we will endeavor to construct a new stack which will allow parsing to continue from that point. Lemma 6 provides a sufficient condition for a new stack to be constructed; we look at the exact conditions for constructing a new stack in Section 3.2.

A method of determining which productions to add to or remove from a grammar must be supplied with the grammar. These productions are encoded in some manner in the portion of the sentence that the parser has already scanned, and must be translated into a form the parser understands. Conceivably, the parser could dictate the manner of this encoding, but we will not allow the parser to do so. Rather, the grammar will include a mechanism for decoding the change to the grammar. This mechanism will take the form of a Turing machine that will take as input the portion of the input already scanned, and will, upon halting, contain upon one of its tapes an encoding of the productions to add to or remove from the grammar in a form that the parser understands; the contents of this tape encode the **grammar transformation** to apply to the grammar. We will let G_T be a context-free grammar with terminal set Σ_T ; grammars and grammar transformations are encoded as strings in $L(G_T)$.

We will generally use the symbol “ Δ ”, or some variation on it, to represent a grammar transformation. Owing to this terminology, we will refer to the Turing machine which produces the transformation as a Δ -machine. An LR(1) grammar, a source sentence, together with a Δ -machine will form the input for our modified parsing algorithm. We will define these objects precisely.

Definition 9. Let Σ be a terminal set. Let M be a Turing machine with three semi-infinite tapes that have tape alphabets Γ_1 , Γ_2 , and Γ_3 , respectively, where $\Gamma_1, \Gamma_2 \supset \Sigma$ and $\Gamma_3 \supset \Sigma_T$. We will require that the machine does not output blanks on the second and third tapes; this allows us to define $w \in \Gamma_2^*$ as the contents of the

second tape up to the first blank cell after the machine halts; similarly, we define $z \in \Gamma_2^*$ for the contents of the third tape up to the first blank cell after the machine halts. If we can guarantee that M halts and that $w \in L(G_T)$, then we say that M is a **Δ -machine over Σ^*** . We define the output of a Δ -machine as (w, Δ) , where Δ is the transformation encoded in z .

Definition 10. A **transformative context-free grammar** (TCF grammar) is a tuple (Σ, N, P, S, T, M) , where:

- Σ is the terminal alphabet,
- N is the nonterminal alphabet,
- P is the production set,
- S is the start symbol,
- T is a subset of P whose elements are the transformative productions of this grammar, and
- M is a Δ -machine over Σ^* .

Definition 11. Let $G = (\Sigma, N, P, S, T, M)$ be a TCF grammar be given, and let $G_0 = (\Sigma, N, P, S)$ be the associated context-free grammar. Let $\alpha, \beta \in (\Sigma \cup N)^*$ be such that

$$\alpha \xrightarrow[G_0]{*} \beta. \quad (2)$$

If no transformative productions are used in (2), then we say that (2) is a **nontransformative derivation**, and we write $\alpha \xrightarrow[G_{\text{nt}}]{*} \beta$. If only transformative productions are used in (2), then we say that (2) is a **transformative derivation**, and we write $\alpha \xrightarrow[G_{\text{t}}]{*} \beta$. We take $\xrightarrow[G]{*}$ to mean $\xrightarrow[G_{\text{nt}}]{*}$.

Definition 12. If $G = (\Sigma, N, P, S, T, M)$ is a TCF grammar, and (Σ, N, P, S) is LR(1), then we say that G is a **transformative LR(1) grammar** (TLR grammar).

Definition 13. A **grammar transformation** for a TLR grammar $G = (\Sigma, N, P, S, T, M)$ is a tuple (N, P_+, P_-) , where:

- N is a set of nonterminals to add to N ;
- P_+ is a set of productions to add to P , where $P \cap P_+ = \emptyset$; and
- P_- is a set of productions to remove from P , where $P_- \subset P$.

This immediately implies that P_- and P_+ are disjoint.

Given a grammar and a transformation, we define the symbol

$$\Delta G \equiv (\Sigma, N \cup N, (P \cup P_+) \setminus P_-, S, T, M).$$

For a given TLR grammar G , let \mathcal{A}_G be the set of all grammar transformations Δ for G .

A couple of observations. Note that T is constant; because $T \subset P$, we must have that $P_- \cap T = \emptyset$. We note also that $\Delta_e = (\emptyset, \emptyset, \emptyset)$ is an identity.

2.4 TLR Languages

At what point should we allow the parser to stop and change the grammar? There are two options: after a shift, or after a reduction. We do not include the option of stopping before a shift because it is not substantively different from stopping after a shift; likewise, we do not include the option of stopping before a reduction. We will adopt the later option: the parser will stop and change the grammar after a reduction.

As a basis for a method of parsing transformative languages, LR(1) parsing would seem to be ideal: LR(1) languages parse from left to right, as is required for transformative languages; LR(1) parsing requires no backtracking; and LR(1) requires single-character lookahead.

Creating an exact definition of transformative LR(1) languages requires the creation of a fair amount of machinery. This will occupy us for the rest of this section. To see why this is work is required, we consider an example in the next section.

2.4.1 A Naive Approach to Transformative Languages

Let us attempt to define the transformative language generated by a TLR grammar in the most obvious way, and see what goes wrong.

Definition 14. If $\alpha \in (\Sigma \cup N)^*$ is a viable prefix for G , and there is some sentential form αax , where $a \in \Sigma$ and $x \in \Sigma^*$, then we say that α is a **viable prefix followed by a** .

Definition 15. Let $G = (\Sigma, N, P, S, T, M)$ and $G' = (\Sigma, N', P', S, T, M)$ be two TLR grammars. Let the alphabets for the three tapes of M be Γ_1, Γ_2 , and Γ_3 , respectively. Let $g \in \Sigma_T^*$ be an encoding of G in $L(G_T)$. Let $\alpha, \beta \in (\Sigma \cup N)^*$ and $x, z \in \Sigma^*$ be given, and let $u, w \in \Gamma_2^*$ also be given. We say that $(\beta z, w, G')$ is a **semiparse** for $(\alpha x, u, G)$, a relationship we denote with the symbol $(\alpha x, u, G) \rightarrow (\beta z, w, G')$, in either of two cases. The first case is that we have all of the following:

1. $\beta z = S$,
2. $\beta z \xrightarrow[G \text{ nt}]{*} \alpha x$,
3. $G' = G$.

The second case is that we have all of the following:

1. $\beta z = \beta' B z \xrightarrow[G \text{ t}]{*} \beta' \gamma z \xrightarrow[G \text{ nt}]{*} \alpha y z = \alpha x$, where $y \in \Sigma^*$ and $\gamma \in (\Sigma \cup N)^*$,
2. β is a viable prefix followed by b for G' , where $b = \text{FIRST}(z)$,
3. the output of M with input (y, u, g) is (w, Δ) such that $G' = \Delta G$.

We could now try to define the language generated by a TLR grammar. Surely, the language generated by a TLR grammar G consists of those strings x such that we have:

$$(x, \epsilon, G) \rightarrow (\alpha_1, u_1, G_1) \rightarrow (\alpha_2, u_2, G_2) \rightarrow \cdots \rightarrow (S, u_n, G_n).$$

As an example of the kind of problem this naive definition of transformative languages poses can be illustrated by example. We let G be the TLR grammar with terminal alphabet $\{c, d\}$, nonterminal alphabet $\{S, A, B, C, D\}$, production set

$$S \rightarrow A \mid B, \quad A \rightarrow C, \quad B \rightarrow D, \quad C \rightarrow c, \quad D \rightarrow d,$$

start symbol S , transformative productions $A \rightarrow C$ and $B \rightarrow D$ and Δ -machine M . Consider the production sets

$$S \rightarrow A, \quad A \rightarrow C, \quad B \rightarrow D, \quad C \rightarrow B, \quad D \rightarrow d, \quad (3)$$

and

$$S \rightarrow B, \quad A \rightarrow C, \quad B \rightarrow D, \quad C \rightarrow c, \quad D \rightarrow A. \quad (4)$$

Let G_1 and G_2 be the grammars that are identical to G , only with the production sets in (3) and (4), respectively. Let Δ_0 , Δ_1 , and Δ_2 be grammar transformations such that

$$\begin{aligned} G_1 &= \Delta_0 G \\ G_2 &= \Delta_1 G_1 \\ G_1 &= \Delta_2 G_2 \end{aligned}$$

We now let the Δ -machine M be such that: when the instantaneous grammar is G , the return value of M is Δ_0 ; when the instantaneous grammar is G_1 , the return value of M is Δ_1 ; and finally, when the instantaneous grammar is G_2 , the return value of M is Δ_2 .

We consider now how to parse the string “ c ” using our naive method:

$$(d, \epsilon, G) \rightarrow (B, u_1, G_1) \rightarrow (A, u_2, G_2) \rightarrow (B, u_3, G_1) \rightarrow (A, u_4, G_2) \cdots.$$

Since the value of u_i does not affect the Δ -machine, for any i , we can see that this sequence of semiparses has no end. This is not a theoretical difficulty, for by Definition 10, we only require that the sequence of semiparses terminates; thus, the string “ c ” is not generated by the grammar G .

Immediately before every transformation of grammar, we have a sentential form which can be derived from the start symbol in the nontransformative grammar associated with the instantaneous grammar in a finite number of steps, say n steps. The problem that arises in this example is that, after the transformation of grammar, the sentential form is now derivable from the start symbol in greater than n steps—in this case, we will say that the parse has been **extended**. Analyzing a given Δ -machine to answer the question of whether or not it will always produce a grammar transformation that will extend the parse is not a task that we can expect the parser to do; indeed, this question is undecidable.

What we can and will do is look for some property of Δ such that, should we require that any transformation emitted by the Δ -machine must have this property, then as a result, the parse will not be extended, hence it can be completed in a finite number of steps.

2.4.2 Allowable Transformations

In this section, we will construct the test the parser can perform to determine if it will accept or reject a transformation emitted by a Δ -machine. In the next section, we shall see if this test does indeed perform as advertised.

The basic idea of the test is to examine the grammar, examine the symbol stack at the time the transformation is to be applied, and identify a certain set of productions called **conserved productions**. The transformation will be considered acceptable if, for each conserved production, there is a corresponding production in the transformed grammar, such that these two productions share a head and a

certain prefix, which we refer to as the **conserved portion** of that production. The function we define now determines if a production is in this set, and if so, how long this prefix is.

For these definitions, we will take $G = (\Sigma, N, P, S, T, M)$ to be given.

Definition 16. Let the sentential form $\beta \in (\Sigma \cup N)^*$ be given, such that $\beta \notin \Sigma^*$; that is, there is at least one nonterminal in β , which we call B . We thus write $\beta = \alpha B a x$, where $\alpha \in (\Sigma \cup N)^*$, $a \in \Sigma$ and $x \in \Sigma^*$; we know that a exists, for at the very least the end-of-file marker appears after the last appearance of B . Let $y \in \Sigma^*$ be such that $\alpha B a x \xrightarrow{*} y$; we consider the parse tree T for $y \in \Sigma^*$. Let B and A be the nodes in T that correspond to the symbols B and a in $\alpha B a x$. Let P be an interior node with n children; we define $N_{\beta, T}(P)$ in one of 4 ways:

1. If P is an ancestor of B , but not an ancestor of A , then let $N_{\beta, T}(P) = n + 1$.
2. If P is an ancestor of A , then one of the children of P is autoancestral to A . Assuming the children of P are X_1, X_2, \dots, X_n , let X_i be the node autoancestral to A ; let $N_{\beta, T}(P) = i$.
3. If P shares an ancestor with both B and A , but is an ancestor of neither, and $B < P < A$, then let $N_{\beta, T}(P) = n + 1$.
4. If none of the above conditions holds, then let $N_{\beta, T}(P) = -1$.

For every $\pi \in P$, we define

$$V_\beta(\pi) = \begin{cases} \max\{N_{\beta, T}(P) : \mathcal{L}(P) = \pi\} & \text{there is some } P \text{ such that } \mathcal{L}(P) = \pi \\ -1 & \text{there is no } P \text{ such that } \mathcal{L}(P) = \pi \end{cases}.$$

We call V_β the **conservation function for β** .

Proposition 1. Let the TLR grammar $G = (\Sigma, N, P, S, T, M)$, the production $\pi \in P$, and the viable prefix β , followed by a , all be given. If $y, z \in \Sigma^*$ are such that both $\beta a y$ and $\beta a z$ are sentential forms for G , then $V_{\beta a y}(\pi) = V_{\beta a z}(\pi)$.

As a result of Proposition 1, we can amend the definition of the conservation function: if β is a viable prefix followed by a , then the value of $V_{\beta a}(\pi)$ is the value of $V_{\beta a x}(\pi)$ for any terminal string x such that $\beta a x$ is a sentential form.

Definition 17. Let the sentential form $\beta \in (\Sigma \cup N)^*$ be given. If $V_\beta(\pi) = -1$, then we say that π is a **free production for β** . If $V_\beta(\pi) = i$, for π equal to $A \rightarrow \beta$, and $0 < i \leq |\beta|$, then we define the first i grammar symbols on the right side of π to be the **conserved production for β** of π . If $V_\beta(\pi) = i$, for π equal to $A \rightarrow \beta$, and $i = |\beta| + 1$, then we define π to be an **entirely conserved production for β** . Entirely conserved productions are also conserved.

Definition 18. Let the transformative grammar $G = (\Sigma, N, P, S, T, M)$, and the sentential form β be given such that $\beta \in (\Sigma \cup N)^*$, but $\beta \notin \Sigma^*$. Let ϕ be a production, not necessarily in P , denoted $B \rightarrow Y_1 Y_2 \dots Y_m$. Let π be a conserved production for β in P . Let π be a production in P , denoted $A \rightarrow X_1 X_2 \dots X_n$. If $V_\beta(\pi) \leq n$, for $1 \leq i \leq V_\beta(\pi)$ we have that $X_i = Y_i$, and $A = B$, then we write $\pi \underset{\beta}{\simeq} \phi$. If $V_\beta(\pi) = n + 1$ and $\pi = \phi$, then we again write $\pi \underset{\beta}{\simeq} \phi$. If P' is a set of productions with terminal set Σ and nonterminal set $N' \supset N$, such that there exists some $\phi \in P'$ for every conserved production $\pi \in P$ satisfying $\pi \underset{\beta}{\simeq} \phi$, then we say that P' **conserves β for P** .

Definition 19. Let the transformative grammar $G = (\Sigma, N, P, S, T, M)$ and the grammar transformation $\Delta \in \mathcal{A}_G$ be given. Let $\Delta G = (\Sigma, N_{\Delta G}, P_{\Delta G}, S, T, M)$. Let αA , a viable prefix followed by a , be given where $\alpha \in (\Sigma \cup N)^*$, $B \in N$, and $a \in \Sigma$. If, for all $x \in \Sigma^*$ such that αAax is a sentential form of G , we have that $P_{\Delta G}$ conserves αAax for P , then we say that Δ is **valid for αBa in G** . The set of all transformations which are valid for αBa we denote as $\mathcal{V}_G(\alpha Ba)$. We include $\Delta_e = (\emptyset, \emptyset, \emptyset)$ in $\mathcal{V}_G(\alpha Ba)$.

To get a feel for how this test accomplishes our goal, we consider the manner in which an LR(1) parser operates. The value on the top of the state stack corresponds to an item set. The item set contains those productions that the parser might be able to reduce. The parser's initial item set is the one containing the item $[S' \rightarrow \cdot S, \cdot]$ and zero or more other items of the form $[A \rightarrow \cdot \alpha, a]$; the parser is in the initial item set only when it has scanned nothing. The parser can either shift or reduce. After the parser shifts a terminal—for example, the terminal “ a ”—the parser goes into a new state, related to the old, as follows: for every item of the form $[A \rightarrow \alpha \cdot a\beta, b]$ in the first state, there is an item of the form $[A \rightarrow \alpha a \cdot \beta, b]$ in the second state. The parser does not stop considering an item just because that item does not have a to the right of the dot; for each item $[A \rightarrow \gamma \cdot B\delta, c]$ that the parser is considering, it looks for an item of the form

$$[D \rightarrow \cdot \zeta, g]; \quad (5)$$

should the parser be able to find a sequence of items like (5), the last of which has the dot to the left of the terminal “ a ”, then the whole sequence of productions remains under consideration.

After the parser reduces a production, say $C \rightarrow \delta$, it pops the appropriate number of states off of the stack, after which, “ C ” will be immediately to the right of the dot in one of the items in the new current state; the parser will move to a new state, in which C is to the left of the dot.

So we can summarize the operation of the parser as follows: the parser considers several production in parallel; for each production, as it shifts terminals, it either:

- advances along that production,
- it records its position in that production, shifting its attention to those productions with the appropriate nonterminal at their head, or
- it drops that production.

This process continues until a reduction, at which time, one of the productions currently under consideration is selected. The parser then recalls its previous state. We want to view the parser stack as the parser's method of keeping track of those productions it is considering. In this light, requiring the transformation to be valid for the current viable prefix means that each production under consideration by the pre-transformation parser has a counterpart that is under consideration by the post-transformation parser—and the portion of the production that the parser has already matched remains unchanged.

The current input symbol is what causes the parser to select the production to use in a reduction; since a grammar transformation takes place immediately after a reduction, the parser will have already “seen” the current input symbol. Say the transformative production just reduced has the nonterminal “ B ” at its head: we know that the current input symbol—say it's “ a ”—is in the follow set of B because

one of the productions the parser had been considering before it matched the transformative production—the production π , for instance—derived some string such that the current input symbol a follows the nonterminal B . Requiring the transformation to be valid for the current viable prefix followed by the current input symbol means that π has a counterpart in the post-transformation grammar that also derives some string such that the current input symbol a follows the nonterminal B , and in this latter derivation is different only inasmuch as it affects parts of the string strictly after the “ a ”.

In other words: any part of the grammar that the parser was using right after the reduction by the transformative production must exist in the new grammar unchanged. Productions not in use, and unused suffixes of productions that were in use, can be modified freely, provided the grammar remains LR(1).

2.4.3 The Language Generated by a TLR Grammar

Since we are interested in those languages for which a parser can be constructed, we found it necessary to restrict those transformations that will be acceptable for a Δ -machine to emit. Now that we have given a precise description of which transformations will be allowed, we can define the concept of a transformative LR(1) language. We begin with Definition 15: the definition of “semiparse”. The following definition is Definition 15, with the requirement that the transformation be valid.

Definition 20. Let $G = (\Sigma, N, P, S, T, M)$ and $G' = (\Sigma, N', P', S, T, M)$ be two TLR grammars. Let the alphabets for the three tapes of M be Γ_1 , Γ_2 , and Γ_3 , respectively. Let $g \in \Sigma_T^*$ be an encoding of G in $L(G_T)$. Let $\alpha, \beta \in (\Sigma \cup N)^*$ and $x, z \in \Sigma^*$ be given, and let $u, w \in \Gamma_2^*$ also be given. We say that $(\beta z, w, G')$ is a **valid semiparse** for $(\alpha x, u, G)$, a relationship we denote with the symbol $(\alpha x, u, G) \rightarrowtail (\beta z, w, G')$, in either of two cases. The first case is that we have all of the following:

1. $\beta z = S$,
2. $\beta z \xrightarrow[G \text{ nt}]{*} \alpha x$,
3. $G' = G$.

The second case is that we have all of the following:

1. $\beta z = \beta' B z \xrightarrow[G \text{ t}]{*} \beta' \gamma z \xrightarrow[G \text{ nt}]{*} \alpha y z = \alpha x$, where $y \in \Sigma^*$ and $\gamma \in (\Sigma \cup N)^*$;
2. β is a viable prefix followed by b for G' , where $b = \text{FIRST}(z)$;
3. the output of M with input (y, u, g) is (w, Δ) such that:
 - (a) $G' = \Delta G$, and
 - (b) $\Delta \in \mathcal{V}_\beta(G)$.

Earlier, we tried to define the language generated by a TLR grammar in the obvious way using the “semiparse” relationship (Section 2.4.1), and we ran into a computational problem. When we define the language generated by a TLR grammar, we do so in much the same way we did before, except that we require the semiparses to be valid.

Definition 21. Let $G_0 = (\Sigma, N, P, S, T, M)$, a TLR grammar, and $x \in \Sigma^*$ be given. Let $w_0 = \epsilon$ and $\alpha_0 = x$. If there is some $k > 0$ such that

$$(\alpha_0, w_0, G_0) \rightarrowtail (\alpha_1, w_1, G_1) \rightarrowtail (\alpha_2, w_2, G_2) \rightarrowtail \cdots \rightarrowtail (\alpha_k, w_k, G_k) = (S, w_k, G_k),$$

then $x \in L(G)$. The set of all strings x is the **language generated by G** .

Theorem 1. *If $L \subset \Sigma^*$ is a recursively enumerable language, then there is a TLR grammar that generates L .*

Proof. We know that there is a Turing machine T that recognizes L . We will construct the TLR grammar $G = (\Sigma, N, P, S, T, M)$ that generates L . Let $\Sigma = \{a_1, a_2, \dots, a_n\}$, let $N = \{S, A, B\}$, let

$$P = \{S \rightarrow A, A \rightarrow B \mid AB, B \rightarrow a_1 \mid a_2 \mid \dots \mid a_n\},$$

and let $T = \{S \rightarrow A\}$. We now describe the operation of M . The input on the first tape of M will be some $x \in \Sigma^*$. If $x \in L$, then output Δ_e ; if $x \notin L$, then output $\Delta = (\emptyset, \emptyset, P)$ —that is, the transformation that removes all productions.

Clearly, if $x \in L$, then $x \in L(G)$, but if $x \notin L$, then $x \notin L(G)$ because the transformation Δ is not valid. \blacksquare

2.5 A Fundamental Theorem of TLR Parsing

We saw before that the basic problem with the naive approach to defining the language generated by a TLR grammar is that the derivation of the string on the symbol stack can be extended. In the last section, we defined the language generated by a TLR grammar using the valid semiparse relation. In this section, we will prove that requiring valid transformations prevents the extension of the string on the symbol stack. This is essential to proving that the TLR parsing algorithm, the subject of the next section, is correct.

We begin with some technical Lemmas, leading up to the main Lemma of this section: Lemma 6. The title theorem is Theorem 2.

Lemma 1. *Let G be a TLR grammar and let αA be a viable prefix followed by c . If $\Delta \in \mathcal{V}_G(\alpha A c)$, and*

$$\beta B c \xrightarrow[G]{*} \alpha A c,$$

then $\Delta \in \mathcal{V}_G(\beta B c)$.

Proof. By contradiction. Let $\Delta G = (\Sigma, N_{\Delta G}, P_{\Delta G}, S, T, M)$. Assume that $\Delta \notin \mathcal{V}_G(\beta B c)$. Thus, there is some $x \in \Sigma^*$ such that $\beta B c x$ is a sentential form, and yet Δ is not valid for $\beta B c x$ in G ; there is some production π that is conserved for $\beta B c x$, where π appears in the derivation

$$S \xrightarrow[G]{*} \beta B c x, \tag{6}$$

such that for no $\phi \in P_{\Delta G}$ do we have $\pi \underset{\beta B c x}{\approx} \phi$.

Isn't π conserved for $\alpha A c x$? Let $y \in \Sigma^*$ such that

$$S \xrightarrow[G]{*} \alpha A c x \xrightarrow[G]{*} y;$$

use the parse tree T_y for y . Let the nodes corresponding to B and c in (6) be B and C , respectively. Let U be the set of all nodes P be a node such that either:

1. P is an ancestor of B , but is not an ancestor of C ;
2. P is an ancestor of C ; or

3. P shares an ancestor with B and C , but is an ancestor of neither, such that $B < P < C$.

These three possibilities correspond to the first three possibilities in Definition 16. Note that

$$S \xrightarrow[G]{*} \beta Bcx \xrightarrow[G]{*} \alpha Acx \xrightarrow[G]{*} y. \quad (7)$$

Let A be the node corresponding to A in the derivation (7). Let W be the set of all nodes Q such that either:

1. Q is an ancestor of A , but is not an ancestor of C ;
2. Q is an ancestor of C ; or
3. Q shares an ancestor with A and C , but is an ancestor of neither, such that $B < Q < C$.

Let $P \in U$. Either B is an ancestor of A , or it shares an ancestor, which we will call T , with both A and C . We go through the three possibilities for P .

1. P is an ancestor of B but not C . There are two ways that this can arise.
 - (a) B is an ancestor of A . This means that P is an ancestor of A but not C , so $P \in W$. Thus, $\mathcal{L}(P)$ is entirely conserved for αAcx .
 - (b) B shares T as an ancestor with A and C , but is an ancestor of neither; thus, P shares T as an ancestor with A and C , but is an ancestor of neither, so $P \in W$, which means that $\mathcal{L}(P)$ is an entirely conserved production for αAcx .
2. P is an ancestor of C , in which case $P \in W$. Thus, $N_{\alpha Acx, T_y}(P) = N_{\beta Bcx}(P)$.
3. P shares an ancestor R with B and C , but is an ancestor of neither. There are two ways that this can arise.
 - (a) B is an ancestor of A . This means that R is an ancestor of B and C , so $P \in W$. Therefore, $\mathcal{L}(P)$ is an entirely conserved production.
 - (b) B shares T as an ancestor with A and C . This means that T is an ancestor of A , P , and C . Therefore, we have that $\mathcal{L}(P)$ is an entirely conserved production

Either way, $N_{\alpha Acx, T_y}(P) = N_{\beta Bcx, T_y}(P)$.

We have here established that $U \subset W$, and that, for all $P \in U$ such that $N_{\beta Bcx, T_y}(P) > 0$, then we have that $N_{\alpha Acx, T_y}(P) = N_{\beta Bcx, T_y}(P)$. It is still possible that $N_{\alpha Acx, T_y} > 0$. Thus we have the following inequality:

$$N_{\beta Bcx, T_y}(P) \leq N_{\alpha Acx, T_y}(P). \quad (8)$$

The fact that $P \in U$, together with (8), lets us conclude that, for any production π that is conserved for βBcx , we have that

$$V_{\beta Bcx}(\pi) \leq V_{\alpha Acx}(\pi). \quad (9)$$

Since we know that $\Delta \in \mathcal{V}_G(\alpha Ac)$, there is some $\phi \in P_{\Delta G}$ such that $\pi \underset{\alpha Acx}{\simeq} \phi$. By (9), we conclude that $\pi \underset{\beta Bcx}{\simeq} \phi$. ■

Lemma 2. *If*

$$\alpha C \xrightarrow[G]{*} \alpha \beta A z \xrightarrow[G]{*} \alpha \beta \gamma B y z$$

and $\Delta \in \mathcal{V}_G(\alpha \beta \gamma B y)$, then

$$\alpha C \xrightarrow[\Delta G]{*} \alpha \beta A x.$$

furthermore, $\text{FIRST}(y) = \text{FIRST}(w)$.

Proof. Assume that $\alpha C \xrightarrow[G]{*} \alpha \beta A z \xrightarrow[G]{*} \alpha \beta \gamma B y z$.

We proceed by induction on the number of steps in the derivation $\alpha C \xrightarrow[G]{*} \alpha \beta A z$.

If $\alpha C \xrightarrow[G]{*} \alpha \beta A z$, then there is a production $C \rightarrow \beta A z$. Therefore, there is a production $C \rightarrow \beta A \delta$ in ΔG ; for any $x \in \Sigma^*$ such that $\delta \xrightarrow[\Delta G]{*} x$, we find that $\alpha C \xrightarrow[\Delta G]{*} \alpha \beta A x$.

Assume this result for all derivations less than n steps long, for $n > 1$. Assume that $\alpha C \xrightarrow[G]{*} \alpha \beta A z$ in n steps. We write

$$\alpha C \xrightarrow[G]{*} \alpha \zeta D \eta \xrightarrow[G]{*} \alpha \zeta D v \xrightarrow[G]{*} \alpha \zeta \theta A u v = \alpha \beta A z.$$

The derivation $D \xrightarrow[G]{*} \theta A u$ is fewer than n steps long, hence the induction hypothesis implies that

$$\alpha' D \xrightarrow[\Delta G]{*} \alpha' \theta A s,$$

where $\alpha' = \alpha \zeta$. Since $\Delta \in \mathcal{V}_G(\alpha \beta \gamma B y)$, there must be a production $C \rightarrow \zeta D \kappa$ in ΔG ; therefore, for any $t \in \Sigma^*$ such that $\kappa \xrightarrow[\Delta G]{*} t$, we have

$$\alpha C \xrightarrow[\Delta G]{*} \alpha \zeta D \kappa \xrightarrow[\Delta G]{*} \alpha \zeta D t \xrightarrow[\Delta G]{*} \alpha \zeta \theta A s t = \alpha \beta A x. \quad \blacksquare$$

Lemma 3. *If*

$$\alpha A \gamma a \xrightarrow[G]{*} \alpha A a, \quad (10)$$

and $\Delta \in \mathcal{V}_G(\alpha A a)$, then

$$\alpha A \gamma a \xrightarrow[\Delta G]{*} \alpha A a.$$

Proof. By induction on the number of steps in the first derivation (10). If $\alpha A \gamma x \xrightarrow[G]{*} \alpha A x$, then there must be a production $B \rightarrow \epsilon$ in G , where $\gamma = B$. Evidently, this production is conserved.

Assume the result for derivations $n \geq 1$ steps long. If there are $n + 1$ steps, then let $\gamma = \delta C$. We know that γ is composed entirely of nonterminals, for any terminals in γ would remain between A and x in the final string of the derivation. We have a production (possibly with ϵ on the right-hand side) $C \rightarrow \zeta$ in G , which is entirely conserved. Thus

$$\alpha A \delta C x \xrightarrow[G]{*} \alpha A \delta \zeta x \xrightarrow[G]{*} \alpha A x.$$

We can use the induction hypothesis to establish that

$$\alpha A \delta \zeta x \xrightarrow[\Delta G]{*} \alpha A x;$$

since $C \rightarrow \zeta$ is also a production in ΔG , we have our result. \blacksquare

Lemma 4. *If G is an TLR grammar, and*

$$\alpha Aa \xrightarrow[G]{*} \beta Ba, \quad (11)$$

and $\Delta \in \mathcal{V}_G(\beta Ba)$, then

$$\alpha Aa \xrightarrow[\Delta G]{*} \beta Ba.$$

Proof. By induction. If $\alpha Aa \xrightarrow[G]{*} \beta Ba$, then there must be some production $A \rightarrow \gamma B$ in G such that $\alpha\gamma = \beta$. In any case, this is a conserved production, so this production is also in ΔG .

Let us assume this Lemma for derivations of length $n \geq 1$ and that there are $n+1$ steps in (11). We can rewrite derivation (11) as

$$\alpha Aa \xrightarrow[G]{*} \alpha \delta C\zeta a \xrightarrow[G]{*} \alpha \delta Ca \xrightarrow[G]{*} \alpha \delta \theta Ba = \beta Ba.$$

We can use Lemma 3 to establish that

$$\alpha \delta C\zeta a \xrightarrow[\Delta G]{*} \alpha \delta Ca.$$

By Case 3 of Definition 16, we see that the production $A \rightarrow \delta C\zeta$ is entirely conserved; thus, $\alpha Aa \xrightarrow[\Delta G]{*} \alpha \delta C\zeta a$. By the induction hypothesis, we have that

$$\alpha \delta Ca \xrightarrow[\Delta G]{*} \beta Ba;$$

therefore,

$$\alpha Aa \xrightarrow[\Delta G]{*} \beta Ba. \quad \blacksquare$$

Lemma 5. *Let G be a TLR grammar. If*

$$\alpha AB \xrightarrow[G]{*} \alpha Aax \quad (12)$$

then

$$\alpha AB \xrightarrow[\Delta G]{*} \alpha Aay,$$

for some $y \in \Sigma^$, provided that $\Delta \in \mathcal{V}_G(\alpha Aa)$.*

Proof. By induction on the number of steps in (12). If $\alpha AB \xrightarrow[G]{*} \alpha Aax$, then there is a production $B \rightarrow ax$ in G ; the conserved portion of this production includes at least the a , therefore, there is a production $B \rightarrow a\beta$ in ΔG . For any $z \in \Sigma^*$ such that $\beta \xrightarrow[\Delta G]{*} z$, we thus have

$$\alpha AB \xrightarrow[\Delta G]{*} \alpha Aa\beta \xrightarrow[\Delta G]{*} \alpha Aaz.$$

Assume the result for all derivations of length not greater than n , for some $n \geq 1$. If there are $n+1$ steps, then let the first production used be $B \rightarrow \beta$.

If the a appears on the right side of this production—that is $\beta = \gamma a\delta$ —then we have

$$\alpha AB \xrightarrow[G]{*} \alpha A\gamma a\delta \xrightarrow[G]{*} \alpha A\gamma ax \xrightarrow[G]{*} \alpha Aax;$$

we can use the preceding Lemma to establish that

$$\alpha A\gamma ax \xrightarrow[\Delta G]{*} \alpha Aax. \quad (13)$$

Since the conserved portion of $B \rightarrow \beta\gamma a\delta$ is at least γa , we must have a production $B \rightarrow \gamma a\zeta$ in ΔG . Thus, for any w such that $\zeta \xrightarrow[\Delta G]{*} w$, we have

$$\alpha AB \xrightarrow[\Delta G]{} \alpha A\gamma a\zeta \xrightarrow[\Delta G]{*} \alpha A\gamma aw \xrightarrow[\Delta G]{*} \alpha Aaw,$$

in light of (13).

If, however, a does not appear on the right side of the production $B \rightarrow \beta$, then there must be some nonterminal C in the conserved portion of the right-hand side of this production deriving the a . That is, there is a production $B \rightarrow \eta C\theta$ in G , such that $\eta C \xrightarrow[G]{*} at$ for some $t \in \Sigma^*$. If $\eta = \epsilon$, then note that

$$\alpha AB \xrightarrow[G]{} \alpha AC\theta \xrightarrow[G]{*} \alpha ACs \xrightarrow[G]{*} \alpha Aats = \alpha Aax;$$

we can apply the induction hypothesis to the derivation $\alpha AC \xrightarrow[G]{*} \alpha Aat$; in this case we have

$$\alpha AB \xrightarrow[\Delta G]{} \alpha AC\theta \xrightarrow[\Delta G]{*} \alpha ACs' \xrightarrow[\Delta G]{*} \alpha Aat's',$$

for appropriate $t', s' \in \Sigma^*$. Now assume that $\eta \neq \epsilon$. Note that

$$\alpha AB \xrightarrow[G]{} \alpha A\eta C\theta \xrightarrow[G]{*} \alpha A\eta Cu \xrightarrow[G]{*} \alpha A\eta\lambda avu \xrightarrow[G]{*} \alpha A\eta avu \xrightarrow[G]{*} \alpha Aavu = \alpha Aax,$$

for appropriate $u, v \in \Sigma^*$. As η is a nonterminal string, let us write $\eta = \eta'E$. Note that

$$\alpha A\eta'ECu \xrightarrow[G]{*} \alpha A\eta'Eavu;$$

in other words,

$$\alpha'EC \xrightarrow[G]{*} \alpha'Eav, \quad (14)$$

where $\alpha' = \alpha A\eta'$. By Lemma 1, $\Delta \in \mathcal{V}_G(\alpha'Ea)$, thus, for every production π in derivation (14), there is some production ϕ such that $\pi \xrightarrow[\alpha'Ea]{*} \phi$. Also, derivation (14) is not more than n steps long. Therefore, we may use the induction hypothesis to yield the derivation

$$\alpha'EC \xrightarrow[\Delta G]{*} \alpha'Eav'.$$

From the previous Lemma, we get

$$AB \xrightarrow[\Delta G]{*} A\eta'ECu' \xrightarrow[\Delta G]{*} A\eta'Eav'u' \xrightarrow[\Delta G]{*} Aav'u'. \quad \blacksquare$$

Definition 22. If $G = (\Sigma, N, P, S)$ is an LR(1) grammar and β is a viable prefix followed by a , then we say that the parser for G will **shift a after n reductions of β** if there is a sequence of $\beta_1, \beta_2, \dots, \beta_n \in (\Sigma \cup N)^*$ such that

$$\beta_n \Rightarrow \dots \beta_2 \Rightarrow \beta_1 \Rightarrow \beta_0 = \beta,$$

and $\beta_n a$ is a viable prefix, but for no $0 \leq k < n$ is it the case that $\beta_k a$ is a viable prefix.

Sometimes it will be useful to use the “converse” of the \Rightarrow symbol.

Definition 23. If $G = (\Sigma, N, P, S)$ is a context-free grammar, and for some $\alpha, \beta \in (\Sigma \cup N)^*$, we have that $\alpha \Rightarrow \beta$, we say that β **reduces to α** , and we write $\beta \xrightarrow{*} \alpha$. Similarly, if $\alpha \Rightarrow \beta$, then we write $\beta \xrightarrow{*} \alpha$.

If G is a transformative grammar, then we give the symbols $\xrightarrow{G \text{ t}}$ and $\xrightarrow{G \text{ nt}}$ the obvious meanings as the converses of the symbols $\xrightarrow{G \text{ t}}$ and $\xrightarrow{G \text{ nt}}$, respectively.

Lemma 6. *Let G be a transformative grammar and let αA be a viable prefix followed by a . If $\Delta \in \mathcal{V}_G(\alpha Aa)$, then αA is a viable prefix followed by a in ΔG , assuming ΔG is TLR.*

Proof. There exists some $x \in \Sigma^*$ such that αAax is a sentential form in G . We will show that there is some $w \in \Sigma^*$ such that αAaw is a sentential form for ΔG . Let $t \in \Sigma^*$ be such that $\alpha Aay \xrightarrow[G]{*} t$, and consider the parse tree for t ; there must be one node P that is an ancestor of both the node representing A and the node representing a , such that the child of P that is autoancestral to the node representing A is different from the child of P that is autoancestral to the node representing a ; let these two children of P be labeled C and X , respectively. Let the production corresponding to P be $P \rightarrow \beta C \gamma X \delta$. Note that

$$X \xrightarrow[G]{*} ay; \quad (15)$$

$$\gamma \xrightarrow[G]{*} \epsilon; \text{ and} \quad (16)$$

$$C \xrightarrow[G]{*} \eta A. \quad (17)$$

We use X to remind us of the possibility that $X = a$; that is, we could have X be either a terminal or a nonterminal.

The derivations in (15), (16), and (17) appear in the derivation for αAax in sequence. Immediately before we begin deriving according to (15), there is a sentential form θPz . We therefore have that $\theta \beta \eta = \alpha$ and $yz = x$.

In light of (17), we have that $C \xrightarrow[\Delta G]{*} \eta A$ by Lemma 4.

In light of (16), we have that $\gamma \xrightarrow[\Delta G]{*} \epsilon$ by Lemma 3.

From the assumption that $\Delta \in \mathcal{V}_G(\alpha Aa)$, we can conclude that there is a production $P \rightarrow \beta C \gamma X \zeta$ in ΔG .

We have two more things to establish: that $X \xrightarrow[\Delta G]{*} au$ for some $u \in \Sigma^*$; and that there is a sentential form θPv for some $v \in \Sigma^*$.

Since

$$S \xrightarrow[G]{*} \theta Pz \xrightarrow[G]{*} \theta \beta \eta Aax,$$

we have, by Lemma 2, that

$$S \xrightarrow[\Delta G]{*} \theta Pv.$$

Therefore, we turn to (15). If $X = a$, then we are done. So assume that X is a nonterminal; let $X = D$. We know that $C \gamma$ ends in a nonterminal string, so let $C \gamma = \lambda E$. By Lemma 1, and Lemma 5, we see that $\Delta \in \mathcal{V}_G(\theta \beta \lambda E a)$. As

$$\theta \beta \lambda E D \xrightarrow[G]{*} \theta \beta \lambda E a y,$$

we therefore have

$$\theta \beta \lambda E D \xrightarrow[\Delta G]{*} \theta \beta \lambda E a w. \quad \blacksquare$$

Theorem 2. *Let G be TLR, let $B \rightarrow \alpha$ be a production in G , and let $\Delta \in \mathcal{V}_G(\gamma Ba)$. If $\gamma \alpha$ is a viable prefix followed by a in G , such that a will be shifted after n reductions of $\gamma \alpha$ in G , then a will be shifted after $n - 1$ reductions of γB in ΔG .*

Proof. By induction on n . If $n = 1$, then the only reduction possible is $\gamma\alpha \xrightarrow[G]{*} \gamma B$. Since a can be shifted, we know that γBa is a viable prefix for G ; by Lemma 6, we see that γBa is a viable prefix for ΔG .

Assume the conclusion for some $k \geq 1$, and assume that $k = n + 1$. Now,

$$\gamma\alpha \xrightarrow[G]{*} \gamma B \xrightarrow[G]{*} \delta.$$

There are n steps in the reduction $\gamma B \xrightarrow[G]{*} \delta$; we can thus write

$$\gamma B = \zeta\eta B \xrightarrow[G]{*} \zeta C \xrightarrow[G]{*} \delta.$$

Note that

$$\zeta Ca \xrightarrow[G]{*} \gamma Ba,$$

therefore, by Lemma 1, we have $\Delta \in \mathcal{V}_G(\zeta Ca)$. By the induction hypothesis, we see that the a can be shifted after $n - 1$ reductions of $\zeta\eta B$ in ΔG . \blacksquare

3 TLR Algorithms

Algorithm 1 (TLR Parsing Algorithm). Parse the string x using the LR parsing algorithm with the parsing table for the CFG grammar associated with G until such a time as a transformative production from G is reduced; at this time, apply a transformation to the grammar, recalculate the parsing tables, and then continue parsing with the new grammar.

Input G : a TLR grammar, where $G = (\Sigma, N, P, S, T, M)$; x : a string over N^*

Output e : a boolean which is true only if $x \in L(G)$

Method

1. Let $w_\Delta = \epsilon$ and $z_\Delta = \epsilon$.
2. Calculate the parse table for G .
3. Push $(0, \epsilon)$ onto the stack.
4. Set a to the first terminal of x .
5. Set s to be the state on the top of the state stack.
6. If $\text{action}[s, a] = \text{shift}$, and $a = \dashv$, then return true.
7. Otherwise, if $\text{action}[s, a] = \text{shift}$, then do the following:
 - (a) Push $(\text{goto}[s, a], a)$ onto the symbol stack.
 - (b) Set $w_\Delta = w_\Delta a$.
 - (c) Set a to the next input symbol.
 - (d) Goto 5.
8. Otherwise, if $\text{action}[s, a] = \text{reduce } \pi$, then do the following (letting $\pi = A \rightarrow \beta$):
 - (a) Pop $|\beta|$ items off the stack.
 - (b) If $\pi \in T$, then execute the Grammar Transformation Algorithm (Algorithm 2); set G and the stack to the returned values.

- (c) Set $w_\Delta = \epsilon$.
- (d) Set s' to be state on the top of the state stack.
- (e) Push $(\text{goto}[s', A], A)$ onto the symbol stack.
- (f) Goto 5.

9. Otherwise, if $\text{action}[s, a] = \text{error}$, then return false. ■

This algorithm closely follows the presentation of the LR parsing algorithm found Section 4.7 of [1]. Indeed, the only essential difference is in Step 8b. We turn now to the previously referenced Grammar Transformation Algorithm.

Algorithm 2 (Grammar Transformation Algorithm). Compute the new grammar and its parsing tables. Assuming the transformation valid, put the parser into the correct state to continue parsing.

Input G : a TLR grammar, where $G = (\Sigma, N, P, S, T, M)$; σ : a parsing stack; w_Δ : a string in Σ^* ; z_Δ : a string in Γ

Output G : a TLR Grammar; σ : a Parser Stack; z_Δ : a string in Γ

Method

1. Execute the Δ -machine with (w_Δ, z_Δ, G) as input, and $(w_\Delta, z_\Delta, \Delta)$ as output.
2. Assert that $\Delta \in \mathcal{V}_G(\alpha)$.
3. Set $G = \Delta G$.
4. Calculate the parsing tables for G .
5. Pop $|\sigma|$ states off of the stack.
6. Do the following until the stack is empty:
 - (a) Let the top item on the stack be (s, X) .
 - (b) Set $\alpha = X\alpha$.
 - (c) Pop the top item off of the stack.
7. Push $(0, \epsilon)$ onto the stack.
8. Do the following, for i from 1 to $|\alpha|$:
 - (a) Let X be the i^{th} symbol of α .
 - (b) Let s be the state on the top of the stack.
 - (c) Push $(\text{goto}[s, a], a)$ onto the stack.
9. Return the new values for G , σ and z_Δ . ■

If $\text{goto}[s, a]$ in Step 8c were ever undefined, then the Grammar Transformation Algorithm fails, which will cause the TLR Parsing Algorithm to fail as well. However, in light of Lemma 6, we can be sure that the Grammar Transformation Algorithm will fail at Step 2 first. It is straightforward to give a useful (to a human) error message in this case.

3.1 Efficiency of The TLR Parsing Algorithm

The efficiency of the Algorithm in the absence of grammar transformations is essentially that of LR parsing. The computation of LR parsing tables is expensive, but since the tables being generated are not wholly independent of the tables that were used up to the point of transformation, an incremental approach is available to us. That is, we need calculate only the portion of the table that has changed.

This idea—incrementally generating parsing tables—was first introduced in the context of an interactive parser generator: the language designer would enter in productions, or modifications to productions, one at a time. After each production was entered, the parser generator would recalculate the affected portion of the parsing tables. As such a system was meant to be interactive, a high premium was placed on response time—hence the development of more efficient algorithms for computing parsing tables.

The two options—a full generation or an incremental generation of parsing tables during a grammar switch—are identical for the consideration of the worst-case performance of a grammar switch operation, because the addition of a single production can cause an exponential increase in the number of parsing states [17].

The TLR parsing algorithm generates canonical LR(1) parsing tables, which are more general, but also far larger, than the more common LALR(1) parsing tables. It is widely quoted (see [1]) that LR(1) parsing tables are much larger than LALR(1) parsing tables. However, LR(1) parsing tables are easy to analyze compared with LALR(1) parsing tables—hence the trade-off. It would be interesting to see how the ideas, algorithms and analysis presented in this work could apply to LALR(1) parsing.

3.2 Correctness

Definition 24. Let $G = (\Sigma, N, P, S)$ be an LR(1) grammar, and let $X_i \in (\Sigma \cup N)$ for $1 \leq i \leq n$. We define $g: (\Sigma \cup N)^* \rightarrow \mathbb{Z}_k$ as follows:

$$g(X_1 X_2 \dots X_n) = \begin{cases} 0 & n = 0 \\ \text{goto}[g(X_1 X_2 \dots X_{n-1}), X_n] & n > 0 \end{cases}.$$

Theorem 3. If α is a viable prefix for G , then $g(\alpha)$ is defined. Furthermore, the items within $I_{g(\alpha)}$ are valid for α .

Proof. Let the viable prefix α be given.

If $|\alpha| = 0$, then $g(\alpha) = 0$. Since the item set containing $[S' \rightarrow \cdot S, \cdot]$ is always I_0 by our convention (established on page 8) and since I_0 is closed, the second conclusion is true in this case.

Assume now that g is defined for all viable prefixes of length not more than n , for some $n \geq 0$, and assume that $|\alpha| = n + 1$; thus, we write $\alpha = \alpha'X$. Let x be a terminal string such that $\alpha'Xx$ is a sentential form. Consider the derivation of $\alpha'Xx$:

$$S' \xrightarrow{*} \beta A_1 z \xrightarrow{*} \beta \gamma X \delta_0 z \xrightarrow{*} \beta \gamma X y z = \alpha' X x. \quad (18)$$

If $\gamma \neq \epsilon$, then it must be true that

$$[B \rightarrow \gamma \cdot X \delta_0, u] \in I_{g(\alpha')}$$

by the assumption that $I_{g(\alpha')}$ contains all valid items for the viable prefix α' .

What if $\gamma = \epsilon$? Clearly, $\beta = \alpha'$. There is a sequence of M steps in (18) that are of the form

$$\alpha' A_{m+1} w_{m+1} v \xrightarrow{*} \alpha' A_m \delta_m w_{m+1} v \quad (19)$$

when going from S' to $\alpha' Xyz$, where we have $A_0 = X$ and $w_0 = y$. Between each step of the form (19), there is a derivation

$$\alpha' A_m \delta_m w_{m+1} v \xrightarrow{*} \alpha' A_m w_m v,$$

for an appropriate $w_m \in \Sigma^*$. If we consider the steps prior to the appearance of $\alpha' A_M w_M z$, we see that

$$S' \xrightarrow{*} \zeta C v \xrightarrow{*} \zeta \eta A_M \delta_M v$$

where $\zeta \eta = \alpha'$. Since M is maximal, we see that $\eta \neq \epsilon$. There is thus an item

$$[C \rightarrow \eta \cdot A_M \delta_M, u] \in I_{g(\alpha')}.$$

Going through our sequence of productions $A_{m+1} \rightarrow A_m \delta_m$ in descending order, we see that

$$[A_M \rightarrow \cdot A_{M-1} \delta_{M-1}, u_{M-1}] \in I_{g(\alpha')};$$

in general

$$[A_{m+1} \rightarrow \cdot A_m \delta_m, u_m] \in I_{g(\alpha')}$$

for all $0 \leq m < M$ because $I_{g(\alpha')}$ is closed.

Consider what we have established: There is an item of the form

$$[Y \rightarrow \gamma' \cdot X \delta', u'] \in I_{g(\alpha')}.$$

Thus, since

$$g(\alpha) = g(\alpha' X) \equiv \text{goto}[g(\alpha'), X],$$

the first conclusion of this Theorem is established.

For the second conclusion of this Theorem, we can use Theorem 5.10 of [2], which justifies the construction of the item sets, and in particular, the item set $I_{g(\alpha' X)}$. ■

It is clear that, in Step 8 of Algorithm 2 calculates g in a bottom-up fashion; it will succeed when that function is defined. Therefore, Theorem 3 gives sufficient condition for the success of that Algorithm.

Definition 25. Let $G = (\Sigma, N, P, S)$ be an LR(1) grammar. Let X_i be a grammar symbol for $1 \leq i \leq n$ such that $X_1 X_2 \dots X_n$ is a viable prefix for G . Define $\mathcal{P}: (\Sigma \cup N)^* \rightarrow (\mathbb{Z}_k^* \times (\Sigma \cup N))^*$ as

$$X_1 X_2 \dots X_n \mapsto ((0, \epsilon), (g(X_1), X_1), (g(X_1 X_2), X_2), \dots, (g(X_1 X_2 \dots X_n), X_n)).$$

Definition 26. Let $G = (\Sigma, N, P, S)$ be an LR(1) grammar. If $\alpha = X_1 X_2 \dots X_n$ is a sentential form for G , then let $1 \leq i \leq n$ such that $X_i \in N$, and $X_{i+1} X_{i+2} \dots X_n \in \Sigma^*$. Let $\beta \equiv X_1 X_2 \dots X_i$ and let $x = X_{i+1} X_{i+2} \dots X_n$. Define the **B-factorization** of α as β and x .

Proposition 2. If G is an LR(1) grammar, and α is a sentential form for G , then the B-factorization of α is unique.

Algorithm 3. We modify Algorithm 1 to include a viable prefix as an input parameter; this viable prefix will be used to initialize the stack. We do this by letting α be the new viable prefix parameter, and we replace Step 3 with

3'. Set the stack to $\mathcal{P}(\alpha)$.

Lemma 7. Let $G_0 = (\Sigma, N, P, S, T, M)$. Let $x \in L(G)$. Let the semiparse sequence for x be

$$(x, u_0, G) \rightarrowtail (\alpha_1, u_1, G_1) \rightarrowtail \cdots \rightarrowtail (\alpha_n, u_n, G_n).$$

For $1 \leq i \leq n$, if α_i is B -factored into β and x , then Algorithm 3 will accept, given β , x , and G_i as input.

Proof. If $n - i = 0$, then $\beta = S$ and $x = \dashv$. By Lemma 3, the prefix parse stack contains all valid items for the form $S \dashv$, which is to say that the item set contains $[S' \rightarrow S \cdot, \dashv]$. Reducing this is an accepting action.

If $n - i > 0$, then we proceed by induction on $n - i$. If we have that $n - i = 1$, then we let $y \in \Sigma^*$ such that $\beta x \xrightarrow[G_i]{*} y$. By Theorem 5.12 of [2], an LR(1) parser will accept y , and at some point during the parsing, the parser will have β on its stack, and x will be its unshifted input. Since

$$(\alpha_i, u_i, G_i) \rightarrowtail (S, u_n, G_n),$$

Algorithm 3 will not apply any grammar transformations; instead, it will execute the same series of actions that an LR(1) parser would once it reaches the aforementioned configuration. Hence, Algorithm 3 will accept.

Assume the result when the input appears as the j^{th} -to-last form in the parse sequence, for some $j > 0$. Assume that $n - i = j + 1$. By Definition 20, we know that there is some α' such that

$$\alpha_i \xrightarrow[G_i]{*} \alpha' \xrightarrow[G_{i+1}]{*} \alpha_{i+1}.$$

By Theorem 5.12 of [2], the parser will correctly trace $\alpha_i \xrightarrow[G_i]{*} \alpha'$, at which point the parser will reduce by a transformative production. This will leave the stack string as the viable prefix γ followed by a ; since the transformation which brings G_i to G_{i+1} is valid for γ , we have by Lemma 6 that γ is a viable prefix for G_{i+1} . Thus, we can use the induction hypothesis to claim that the parser will accept γ . ■

Lemma 8. Let $G = (\Sigma, N, P, S, T, M)$ be a TLR grammar. If α is a viable prefix followed by a , and an LR(1) parser for the LR(1) grammar (Σ, N, P, S) would shift a after n reductions if α is on the parsing stack as a is the lookahead, then an TLR parser will shift a after n reductions if α is on the parsing stack as a is the lookahead

Proof. By induction on n . If $n = 0$, then both parsers will immediately shift a .

Assume the result for some $k \geq 0$, and assume that $n = k + 1$. Since both of the parsers will initially have identical stacks, they will reduce by the same production. If it happens that this production is not transformative, then the parsers will have identical stacks after the first reduction, after which we can apply the induction hypothesis. If it happens that this production is transformative, then, letting the new grammar by ΔG , we can apply the induction hypothesis by virtue of Theorem 2. ■

Theorem 4. Let $G = (\Sigma, N, P, S, T, M)$ be a TLR grammar. The TLR Parsing Algorithm (1) recognizes $L(G)$.

Proof. Let $x \in \Sigma^*$.

If $x \in L(G)$, then, because Algorithm 3 operates as Algorithm 1 does when $\alpha = \epsilon$, we know that the parser will accept x , given Lemma 7.

Assume, then, that $x \notin L(G)$. There are several ways in which this could happen. First, the Δ -machine could emit a transformation that is not valid; if this happens, then the parser will clearly reject x . Second, after a shift or a reduction by a production that is not transformative, it could be that the stack string is not a viable prefix, or the lookahead might not follow the stack string; in either case, by Theorem 5.12 of [2], the parsing tables will call for an error action, and so the parser will reject the string.

The only other possibility is that there is a sequence of tuples

$$(x, u_0, G) = (\alpha_0, u_0, G_0) \rightarrowtail (\alpha_1, u_1, G_1) \rightarrowtail (\alpha_2, u_2, G_2) \rightarrowtail \dots$$

with no upper bound on the length of this sequence. We shall dispose of this possibility presently. Assume that such a sequence exists. Choose some $i \geq 0$, and let $\alpha_i = \beta B y$, where $\beta \in (\Sigma \cup N)^*$, $B \in N$, and $y \in \Sigma^*$. Note that $\beta B \equiv \gamma$ is a viable prefix. We proceed by induction on $|y|$. If $|y| = 1$, then we note that an LR(1) parser would shift the first symbol of y (specifically: \rightarrow) after m reductions. By Lemma 8, the TLR parser will shift the first symbol of y after m reductions. Assume that the parse always terminates for strings of length $k \geq 1$, and assume that $|y| = k + 1$. The parser will, in light of Lemma 8, shift the first symbol of y after a finite number of reductions. At this point, we have a viable prefix, followed by a k -character string, allowing us to apply the induction hypothesis. Therefore, the parse always completes.

Since we have exhausted the possible reasons why $x \notin L(G)$, we conclude that the parser recognizes $L(G)$. ■

4 Checking the Validity of a Transformation

In the previous section, we considered the set of valid transformations for a given viable prefix and lookahead symbol. In this section, we develop an algorithm to determine if a particular transformation is valid, and we provide a correctness proof of the same.

4.1 Computing the Conservation Function

We have discussed the criteria for membership of a transformation in the set of valid transformations; these criteria must be met by transformations emitted by the Δ -machine. It is not immediately clear how we are to determine whether or not a transformation is in this set. We consider a method of making this determination presently.

Algorithm 4 (An Algorithm to Compute the Conservation Function). This algorithm computes a conservation function much like $V_\beta(\pi)$. It is straightforward to test the transformation for validity, given this function. The construction of the set is accomplished by tracing all of the different ways we might decide that the lookahead gets parsed from the start symbol. As we trace through the different productions in the grammar, we record our progress in sets of ordered pairs. The inclusion of an ordered pair like (π, k) in one of these sets, labeled $V_{\text{something}}$, means

that one of the procedures invoked during the execution of this Algorithm visited the first k symbols of π ; fortuitously, this turns out to be exactly what we need to generate the conservation function.

Input $G = (\Sigma, N, P, S, T, M)$: a TLR grammar; $\sigma = ((s_0, \epsilon), (s_1, X_1), \dots, (s_m, X_m))$: a parse stack; a : a terminal called the “lookahead”

Output V_P : a set of ordered pairs $(C \rightarrow \delta, i)$, where $C \rightarrow \delta \in P$, and $i \leq |\delta|$

Method

1. Calculate the item sets for G ; let them be I_0, I_1, \dots, I_p .
2. Let V_T be an empty set of ordered pairs of the same type as V_P .
3. Let (s, B) be the item on the top of the stack.
4. For every item of the form $j = [A \rightarrow \alpha B \cdot \gamma, b]$ in I_s , do the following:
 - (a) Call Procedure 5 with G , the stack, j , and a as input; let V_F and f be its output.
 - (b) Set $V_T = V_T \cup V_F$.
 - (c) If f is true, then call Procedure 6 with G , the stack, j , and a as input; let V_A be its output, and set $V_T = V_T \cup V_A$.
5. For every production, $\pi \in P$, define V_x as follows:

$$V_x(\pi) = \begin{cases} \max\{i \in \mathbb{Z} : (\pi, i) \in V_T\} & \text{there exists some such } i \\ -1 & \text{otherwise} \end{cases}.$$

6. Let $V_P = \{(\pi, V_x(\pi)) : \pi \in P\}$. Return V_P . ■

Procedure 5. This procedure starts from an item in the current item set and searches for all of the ways that the lookahead could be included by that item, if we assume that the production in the given item must eventually be reduced. It does this by considering γ , the “tail” of the item in question. Each symbol of γ is considered, continuing as long as ϵ can be derived from the current symbol, until ϵ cannot be derived. If it turns out that $\gamma \xrightarrow{*} \epsilon$, then we back up in the symbol stack to where the parser first started to consider the current item, and we recursively retry this Procedure from that point. Upon halting, we return V_F and f . The set of ordered pairs V_F records which productions we have visited during the execution of this Procedure, or one of the procedures invoked during its execution. The flag f indicates whether we found any way of deriving the lookahead.

Input $G = (\Sigma, N, P, S, T, M)$: a TLR grammar; $\sigma = ((s_0, \epsilon), (s_1, X_1), \dots, (s_m, X_m))$: a parse stack; a , a terminal; $j = [A \rightarrow \alpha B \cdot \gamma, b]$: an item

Output V_F : a set of ordered pairs $(C \rightarrow \delta, i)$, where $C \rightarrow \delta \in P$, and $i \leq |\delta|$; f : a boolean flag

Method

1. Calculate the item sets for G ; let them be I_0, I_1, \dots, I_p .
2. Set f to false.
3. Let V_Z be an empty set of ordered pairs, of the same type as V_F .
4. Let $\gamma = Y_1 Y_2 \dots Y_n$.

5. Let π be the production $A \rightarrow aB\gamma$.
6. Let i range from 1 to n , and do the following:
 - (a) If $Y_i = a$, then add $(\pi, |\alpha| + 1 + i)$ and the contents of V_Z to V_F .
 - (b) If Y_i is a terminal, then return V_F .
 - (c) If Y_i is a nonterminal, then call Procedure 7 with G , Y_i , a , and \emptyset as the input; let the output be V_E , f_N , and e (we ignore Π).
 - (d) If e is true or f_N is true, then set $V_Z = V_Z \cup V_E$.
 - (e) If f_N is true, then add $(\pi, |\alpha| + 1 + i)$ and the contents of V_Z to V_F , set $V_Z = \emptyset$ and set f to true.
 - (f) If e is false, then return V_F and f .
7. Add $(\pi, |\alpha B\gamma| + 1)$ to V_Z .
8. Pop $|\alpha| + 1$ items off of the stack. Let s be the state in the top item of the stack.
9. Set $J = \{j_0\}$, where j_0 is the item $[A \rightarrow \cdot\alpha B\gamma, b]$ that is in I_s .
10. Repeat the following until no more items can be added to J .
 - (a) If there is an item of the form $[C \rightarrow \cdot\delta, c]$ in J and an item of the form $[D \rightarrow \zeta \cdot C\eta, d]$ in I_s , then add the latter item to J , if it is not already in J .
11. For each item $[E \rightarrow \theta \cdot \kappa, e]$ in J , do the following:
 - (a) Let ϕ be the production $E \rightarrow \theta\kappa$.
 - (b) Add the ordered pair $(\phi, |\theta| + 1)$ to V_Z .
 - (c) Call this Algorithm recursively with G , a , and the current value of the stack as input, along with $[E \rightarrow \theta \cdot \kappa, e]$ in place of j ; let V_P and f_P be the output.
 - (d) If f_P is true, then set f to true, set $V_Z = V_Z \cup V_P$, and set $V_Z = \emptyset$.
12. Return V_F and f . ■

Procedure 6. This procedure takes an item j in the current item set, and finds all of the items in one of the preceding item sets that might be reduced, if we assume that j must be reduced. These “ancestor” items of our item j do not need to be totally conserved: they only need those symbols to the left of and immediately to the right of the dot to be conserved. Once this procedure has popped some item sets off of the stack, then it calls itself recursively.

Input $G = (\Sigma, N, P, S, T, M)$: a TLR grammar; $\sigma = ((s_0, \epsilon), (s_1, X_1), \dots, (s_m, X_m))$: a parse stack; a , a terminal called the “lookahead”; $j = [A \rightarrow \alpha B \cdot \gamma, b]$: an item

Output V_A : a set of ordered pairs $(C \rightarrow \delta, i)$, where $C \rightarrow \delta \in P$, and $i \leq |\delta|$

Method

1. Calculate the item sets for G ; let them be I_0, I_1, \dots, I_p .
2. Pop $|\alpha| + 1$ items off of the stack. Let s be the state in the element on the top of the symbol stack.
3. Let $J = \{j\}$.
4. Repeat the following until no more items can be added to J .

- (a) If there is an item of the form $[C \rightarrow \cdot\delta, c]$ in J and an item of the form $[D \rightarrow \cdot C\zeta, d]$ in I_s , then add the latter item to J , if it is not already in J .
5. For every item $k = [E \rightarrow \eta \cdot \theta, d]$ in J , do the following:
 - (a) Let π be the production $E \rightarrow \eta\theta$.
 - (b) Add $(\pi, |\eta| + 1)$ to V_A .
 - (c) If $\eta \neq \epsilon$, then call this Algorithm recursively with G and a , the current values of the stacks as input, along with k in place of j ; let the output be V'_A .
 - (d) Set $V_A = V_A \cup V'_A$. ■
6. Return V_A .

Procedure 7. Determine all of the ways that a given grammar symbol can derive the lookahead. If the grammar symbol is a terminal, then do nothing. If the grammar symbol is a nonterminal, then return V_S , a set representing the portions of the productions that we visited during the execution of this and the following Procedure. This procedure calls itself recursively, so care must be taken if we are dealing to avoid an infinite loop; we use Π , a set of the productions that this Procedure has already visited, that is both input to and output from this Procedure. We also return two flags: the flag e is true if and only if $X \xrightarrow{*} \epsilon$; the flag f is true if and only if $X \xrightarrow{*} ax$, for some $x \in \Sigma^*$.

Input $G = (\Sigma, N, P, S, T, M)$: a TLR grammar; X : a symbol in $\Sigma \cup N$; Π : a set of productions; a : a terminal

Output V_S : a set of ordered pairs (π, n) ; Π : a set of productions; e : a boolean flag; f : a boolean flag

Method

1. Set both e and f to false.
2. If X is a terminal, then do the following:
 - (a) Set $V_S = \emptyset$.
 - (b) If $X = a$, then set f to true.
 - (c) Return V_S, Π, f and e .
3. For every production $X \rightarrow \alpha$, such that $X \rightarrow \alpha \notin \Pi$, do the following:
 - (a) Add $X \rightarrow \alpha$ to Π .
 - (b) Execute Procedure 8 with G, α, Π , and a as input, and let V_*, Π_*, f_* and k be the output.
 - (c) If $k = |\alpha| + 1$, then set e to true.
 - (d) Set $\Pi = \Pi \cup \Pi_*$.
 - (e) If f_* is true, then set f to true.
 - (f) If f_* is true or e is true, then set $V_S = V_S \cup V_*$, and put (π, k) in V_S . ■
4. Return V_S, Π, f and e .

Procedure 8. Determine all of the ways that a given string of grammar symbols can derives a string beginning with the lookahead. Return this information in V_* , a set representing the portions of the productions that we visited during the execution of this and the preceding Procedure. The set Π is used for the same purpose as in

Procedure 7. The integer k encodes the result of this Procedure's execution as follows: if $\gamma \xrightarrow{*} \epsilon$, then we return $k = |\gamma| + 1$; otherwise, if $\gamma \xrightarrow{*} ax$, for some $x \in \Sigma^*$, then we let $1 \leq k \leq |\gamma|$; otherwise, we let $k = -1$. If we have both that $\gamma \xrightarrow{*} \epsilon$ and that $\gamma \xrightarrow{*} ay$, for some $y \in \Sigma^*$, then we let $k = |\gamma| + 1$ and we let f be true.

Input $G = (\Sigma, N, P, S, T, M)$: a TLR grammar; γ : a string over $(\Sigma \cup N)^*$; Π : a set of productions; a : a terminal

Output V_* : a set of ordered pairs (π, n) ; Π : a set of productions; f : a boolean flag; k : an integer with $-1 \leq k \leq |\gamma| + 1$

Method

1. Set $k = -1$.
2. Set f to false.
3. Let $\gamma = X_1 X_2 \dots X_n$.
4. For each i from 1 to n , do the following:
 - (a) Execute Procedure 7 with G , X_i , and Π as input and V_S , Π_S , e , and f_S as output.
 - (b) Set $\Pi = \Pi \cup \Pi_S$.
 - (c) If f_S is true or e is true, then set $k = i$.
 - (d) If f_S is true, then set f to true.
 - (e) Set $V_* = V_* \cup V_S$.
 - (f) If e is false, then go to Step 6.
5. Return V_* , Π , f and $n + 1$.
6. If f is false, then set $V_* = \emptyset$ and set $k = -1$.
7. Return V_* , Π , f and k . ■

In Procedure 5, we call Procedure 7 in Step 6c. Since calling Procedure 7 twice with the same nonterminal does not yield any new information, as an optimization, we could keep track of the nonterminals that have already been passed to Procedure 7, calling that Procedure only if we have not called it with that nonterminal before. As an additional optimization, we could retain Π in the same step, and not pass \emptyset to Procedure 7. We leave the algorithm as it is because it makes it a little easier to analyze, a task which we turn to now.

4.1.1 A Model of the Operation of the Algorithm

We will endeavor to prove that the Algorithm is correct. This will be exceedingly dull and difficult if we attempt to do so directly. Rather, we will model the operation of the algorithm in simple, formal terms in this section. With this model in hand, it will be possible to produce the desired proof. Note that we do not formally assert the equivalence of the Algorithm presented in this section and the model presented here: we will, however, take the “model” to be authoritative.

Definition 27. Let the transformative context-free grammar $G = (\Sigma, N, P, S, T, M)$ be given such that the context-free grammar (Σ, N, P, S) is $LR(k)$. Let the item set I be given. If $i = [A \rightarrow \beta, y]$ and $j = [C \rightarrow \gamma \cdot A\delta, z]$ are two items in I , then we write $j \triangleright_k i$.

Definition 28. Let the transformative context-free grammar $G = (\Sigma, N, P, S, T, M)$ be given, such that the context-free grammar (Σ, N, P, S) is $LR(k)$. Let the collection of item sets for G be I_0, I_1, \dots, I_n . Finally, let the parser stack

$$((s_0, \epsilon), (s_1, X_1), \dots, (s_m, X_m))$$

be given. Let i and j be two items, and let $1 \leq p \leq m$ such that $i \in I_{s_p}$. If either:

1. $j \triangleright_k i$; or
2. j is in $I_{s_{p-1}}$ and it is of the form $[A \rightarrow \alpha \cdot X_p \beta, z]$, such that i is of the form $[X_p \rightarrow \cdot \gamma, y]$;

then we write $j \blacktriangleright_k i$.

Definition 29. Let the transformative context-free grammar $G = (\Sigma, N, P, S, T, M)$ be given, such that the context-free grammar (Σ, N, P, S) is $LR(k)$. Let the collection of item sets for G be I_0, I_1, \dots, I_n . Finally, let the parser stack

$$((s_0, \epsilon), (s_1, X_1), \dots, (s_m, X_m))$$

be given. Let j_1, j_2, \dots, j_q be a sequence of items, and let $\rho_1, \rho_2, \dots, \rho_q \in \{0, 1, \dots, n\}$, such that, for $1 \leq i < q$, we have that $\rho_{i+1} - \rho_i$ is 0 or 1. If we have all of the following, then we say that $(j_1, j_2, \dots, j_q; \rho_1, \rho_2, \dots, \rho_q)$ is a k -parse precession:

1. $j_q \in I_s$, where $s = s_{\rho_q}$;
2. for $1 \leq r < q$,

$$j_r \blacktriangleright_k j_{r+1};$$

3. j_1 is of the form $[C \rightarrow \cdot \gamma, x]$, where $|x| = k$; and

4. there do not exist indices p and q such that $j_p = j_q$ and $\rho_p = \rho_q$, and for all $p \leq h < q$ we have $j_h \triangleright_k j_{h+1}$ and for some $p < r < q$, we have that either $j_p = j_r$ or that $j_q = j_r$.

We often omit the second component of a k -parse precession—namely, the sequence of indices $\rho_1, \rho_2, \dots, \rho_q$ —unless they are explicitly called for.

Definition 30. Let $G = (\Sigma, N, P, S, T, M)$ be a TLR grammar, let $\beta B = X_1 X_2 \dots X_p$ be a viable prefix followed by a , and let $((s_0, \epsilon), (s_1, X_1), \dots, (s_p, X_p))$ be a parse stack. Let $J = (j_1, j_2, \dots, j_n; \rho_1, \rho_2, \dots, \rho_n)$ be a 0-parse precession, let $U = (u_1, u_2, \dots, u_m; \tau_1, \tau_2, \dots, \tau_m)$ be a 1-parse precession, and let $s = [A \rightarrow \delta \cdot \gamma]$ be an $LR(0)$ item. Let $u_i = [C_i \rightarrow \zeta_i \cdot \eta_i, a]$, for $1 \leq i \leq m$. Consider the following criteria:

1. $j_1 = [S' \rightarrow \cdot S]$;
2. either of the following:
 - (a) $m = 0$, in which case, s is of the form $[A \rightarrow \delta' B \cdot \gamma]$ where $\delta' B = \delta$, and we have that $j_n \blacktriangleright s$ and that $\rho_n = p - 1$, or
 - (b) $m > 0$, and all of the following:
 - s is of the form $[A \rightarrow \delta' C \cdot \gamma]$ for $\delta' C = \delta$,
 - u_m is of the form $[D \rightarrow \theta B \cdot \kappa, a]$,
 - $\tau_m = p$;
3. $\gamma \xrightarrow{*} ax$, for some $x \in \Sigma^*$;

4. all $\eta_h \xrightarrow{*} \epsilon$, for $h \geq 1$.

If all of these criteria hold, then we say that U , J , and s constitute an **upward link** to a , an **ancestral link**, and a **sidelink** to a , respectively for βB , and we say that U and J **join** s .

Definition 31. Let βB be a viable prefix. Let $U = (u_1, u_2, \dots, u_n)$ be a 1-parse precession. Let $P = \{p_1, p_2, \dots, p_m\}$ be the set of indices such that $p_1 < p_2 < \dots < p_m$ and for $1 \leq h \leq m$, we have that j_{p_h} is of the form $[A \rightarrow \alpha \cdot \gamma]$ where $\alpha \neq \epsilon$, and we also have that, for any index $h_0 \notin P$, we have that j_{h_0} is of the form $[A \rightarrow \cdot \gamma]$. We let $j_{p_h} = [A_h \rightarrow \alpha_h X_h \cdot \gamma_h]$ for $1 \leq h \leq n_p$, where X_h is a grammar symbol. Let $\delta \equiv X_1 X_2 \dots X_m$; we call δ the **trace** of U .

Definition 32. Let βB be a viable prefix followed by a . Let U, J be an upward and an ancestral link, joining the sidelink s . Let δ_J be the trace of J . There are two cases that we will consider:

1. if $|U| = 0$, then we know that s is of the form $[A \rightarrow \alpha B \cdot \gamma]$. Let $\delta \equiv \delta_J B$; otherwise,
2. if $|U| > 0$, then we let $\delta \equiv \delta_J \delta_U$, where δ_U is the trace of U .

We call δ the **trace** of the tuple (U, J, s) .

Theorem 5. Let βB be a viable prefix followed by a . Let U, J be an upward and an ancestral link, joining the sidelink s . The trace of (U, J, s) is βB .

Proof. Let $\sigma = (s_0, s_1, s_2, \dots, s_n)$ be the state stack, and let $I_0, I_1, I_2, \dots, I_m$ be the item sets for G . Let $U = (u_1, u_2, \dots, u_p; \tau_1, \tau_2, \dots, \tau_p)$ be a 1-parse precession, and let $J = (j_1, j_2, \dots, j_q; \rho_1, \rho_2, \dots, \rho_q)$ be a 0-parse precession. Let $\beta = X_1 X_2 \dots X_n$.

We will first define the LR(0) item r :

1. if $|U| = 0$, then let $r = s$;
2. if $|U| > 0$, then let $u_1 = [A_U \rightarrow \alpha_U \cdot \gamma_U, a]$ and let $r = [A_U \rightarrow \alpha_U \cdot \gamma_U]$.

We begin by proving that the trace of J is a prefix of βB that is of length ρ_q . We shall proceed by induction on ρ_q . Assume that $\rho_q = 0$. For all $j \in J$, we know that j is of the form $[A_j \rightarrow \cdot \alpha_j]$. Thus, the trace of J is ϵ .

Assume that we know that the trace of J is a prefix of βB that is ρ_q symbols long when $|J| = n_j$, where $n_j \geq 0$. Let us assume that $\rho_q = n_j + 1$ symbols long. Let q_g be the greatest index such that $\rho_{q_g} = n_j$. We know that $\rho_{q_g} = n_j + 1$. Thus,

$$j_{q_g} \triangleright j_{q_g+1} \quad \text{but} \quad j_{q_g} \not\triangleright j_{q_g+1}.$$

Since j_{q_g+1} is of the form $[A_g \rightarrow \alpha_g X_g \cdot \gamma_g]$, we must have, by Condition 2 of Definition 28, that

$$X_g = X_{n_j+1}.$$

Since we know, by the induction hypothesis, that the trace of $J' = (j_1, j_2, \dots, j_{q_g})$ is

$$X_1 X_2 \dots X_{n_j},$$

we conclude that the trace of J is

$$X_1 X_2 \dots X_{n_j+1}.$$

We now consider the possibility that $|U| = 0$. In this case, we have—by Condition 2a of Definition 30—that $\rho_q = n - 1$. Also, we know that s is of the form

$$[A_s \rightarrow \delta_s B \cdot \gamma_s].$$

Thus, the trace of J is β , so by Condition 1 of Definition 32, we see that the trace of (U, J, s) is βB .

If $|U| > 0$, then note that

$$\tau_1 = \rho_q + 1.$$

Let $t = \tau_p - \rho_q$. We shall proceed by induction on t .

Assume that $t = 1$. Now, as we know that $\tau_h = \rho_q + 1$ for all $1 \leq h \leq p$, we conclude that u_1 is of the form

$$[C_u \rightarrow \zeta_u X_u \cdot \eta_u, a]$$

and that when $1 < h \leq p$, we have that u_h is of the form

$$[C_h \rightarrow \cdot \theta_h, a].$$

However, we know that u_p is of the form

$$[C_* \rightarrow \zeta_* B \cdot \eta_*, a].$$

Therefore, we note that $|U| = 1$, and by Condition 2b of Definition 30, we note that $s(j_q) = n - 1$. Thus, the trace of U is B , and since the trace of J is β , we conclude that the trace of (U, J, s) is βB .

Assume now that the trace of $|U|$ is known to be

$$X_{n-t+2} X_{n-t+3} \dots X_n B$$

when $t = k$, for $k \geq 1$. Assume that $t = k + 1$. Consider u_1 : let $u_1 = [A_{u_1} \rightarrow \alpha_{u_1} \cdot \gamma_{u_1}, a]$; now let $u_f = [A_{u_1} \rightarrow \alpha_{u_1} \cdot \gamma_{u_1}]$. We know that

$$j_p \triangleright u_f \quad \text{but} \quad j_p \not\triangleright u_f;$$

thus, $s(j_p) = s(u_1) - 1$. Since $j_p \not\triangleright u_f$, we conclude that u_1 is of the form $[A_{u_1} \rightarrow \alpha'_{u_1} X_{u_1} \cdot \gamma_{u_1}, a]$, where $\alpha'_{u_1} X_{u_1} = \alpha_{u_1}$. As u_1 is a live item in I_{τ_1} , we conclude that $X_{u_1} = X_{n-t+2}$. Therefore, by induction, the trace of U is

$$X_{n-t+2} X_{n-t+3} \dots X_n B.$$

We now know that the trace of J is $\beta_J = X_1 X_2 \dots X_{\rho_q}$. Also, when we assume that $|U| > 0$, we also know that the trace of U is $\beta_U = X_{\tau_1} X_{\tau_2} \dots X_{\tau_{p-1}} B$. Since $\tau_1 = \rho_q + 1$, and since $\tau_p = n$, we have therefore established that $\beta_J \beta_U = \beta B$. ■

Definition 33. Let $G = (\Sigma, N, P, S, T, M)$ be a TLR grammar. Let a and γ be given, such that $\gamma \xrightarrow{*} ax$, where $x \in \Sigma^*$. Let $\gamma = X_1 X_2 \dots X_m$. If k is an index such that

$$X_1 X_2 \dots X_{k-1} \xrightarrow{*} \epsilon \tag{20}$$

and

$$X_k \xrightarrow{*} ay,$$

where $y \in \Sigma^*$, then (k, γ) comprises a **partial downward link to a for γ** . Of course, if $k = 1$, then we take the symbol $X_1 X_2 \dots X_{k-1}$ to be a synonym for ϵ , in which case (20) is trivial.

Definition 34. Let $G = (\Sigma, N, P, S, T, M)$ be a TLR grammar. Let a and γ be given, such that $\gamma \xrightarrow{*} ax$, where $x \in \Sigma^*$. Let $\gamma = X_1 X_2 \dots X_m$. If k is an index such that

$$X_1 X_2 \dots X_{k-1} \xrightarrow{*} \epsilon$$

and $X_k = a$, then we say that (k, γ) is a **terminal link to a for γ** .

Definition 35. Let L be a partial downward link to a for γ , with value k . Let $\gamma = X_1 X_2 \dots X_n$. If L_d is a partial downward link to a for γ_d , then L_d is **chained to L** if $X_k \rightarrow \gamma_d$ is a production. If L_t is a terminal link to a for γ_t , then L_t is **chained to L** if $X_k \rightarrow \gamma_t$ is a production. If L_1 and L_2 are two links, then we define the **chain production** to be either $X_k \rightarrow \gamma_d$ or $X_k \rightarrow \gamma_t$, as appropriate, and we represent this production with the symbol $P(L_1, L_2)$.

Definition 36. Let L_1, L_2, \dots, L_n be a sequence of chained links, where for each $L_i = (\gamma_i, a)$ such that (γ_i, k_i) is a partial downward link for γ_i to a when $i < n$, while (γ_i, k_i) is a terminal link for γ_i to a when $i = n$. If, for $1 \leq j < n$, there is most one other index $1 \leq k < n$ such that $j \neq k$ but $P(L_j, L_{j+1}) = P(L_k, L_{k+1})$, then we say that this sequence of strings, partial downward links and this terminal link comprises a **complete downward link from γ_1 to a** .

Definition 37. Let $G = (\Sigma, N, P, S, T, M)$ be a TLR grammar, and let β be a viable prefix followed by a . Let U, J be an upward and an ancestral link, joining the sidelink s . Letting $s = [A \rightarrow \delta \cdot \gamma]$, let the complete downward link D from γ to a be given. Let $U = (u_1, u_2, \dots, u_n; \tau_1, \tau_2, \dots, \tau_n)$ such that, for $1 \leq i \leq n$, we have u_i of the form

$$u_i = [C_i \rightarrow \alpha_i \cdot \delta_i, a].$$

If we have that such that $\delta_i \xrightarrow{*} \epsilon$ for $1 \leq i \leq n$, then we call the ordered quadruple (U, J, s, D) a **parse path for the viable prefix β followed by a** .

We will argue that Algorithm 4 operates by enumerating all parse paths for the viable prefix β followed by a .

Let us say that we have a TLR grammar $G = (\Sigma, N, P, S, T, M)$, and that we are parsing a sentence x . The parser has just reduced by a transformative production, leaving the stack as βB , with lookahead a . Algorithm 4 begins with all of the items in the item set on the top of the stack that are of the form $[A \rightarrow \beta B \cdot \gamma, b]$, which is incidentally the form of the sidelink. The Algorithm's next step is to invoke Procedure 5 to find the downlinks.

Procedure 5 scans the “remainder” of the current item—that is, the portion to the right of the dot—to determine if this remainder can be used to derive a or ϵ . The way that it makes this determination is with Procedures 7 and 8. If these Procedures successfully find such a derivation, then they return the portions of each production that they used in V_S and V_* ; if they are not successful, then those two sets are empty. If these Procedures determine that the remainder derives ϵ , then Procedure 5 will find all items that preceeds the current item in the parse precession. The way that these items are found is by first “rewinding” the parse stack until such a time as the parse first started to consider the current item. At this point, we create J , which is like the closure of an item set, taken in reverse. We consider the items in J one at a time. For a parse precession, we do not allow an item to be present more than twice, unless we go to the previous item set; in Procedure 5, we create J first, then we call the Procedure recursively for each item in the set. Since, in Step 8 of the Procedure, we pop at least one item off of the

stack, and since we call the Procedure recursively exactly once for each item in J , we can be sure that the sequence of items we trace does not violate condition 4 of Definition 29.

We do not allow a production to appear more than twice in the complete downward link to a , so we use the production set Π , which we initialize to \emptyset when we invoke Procedure 7 in Procedure 5.

During this process, each of the upward links to a are enumerated, as are the complete downward links to a , with the appropriate sidelink for one of the parse precessions.

As for the ancestral links, we have Procedure 6, which is invoked only if Procedure 5 succeeds in finding a way to derive a from an item.

What of V_P , the set returned by Algorithm 4? We have gone to some effort to construct a model of the operation of Algorithm 4, but we have no counterpart for the set V_P . We now construct a function which, given a parse path and a production π , returns the value of $V_P(\pi)$ that the Algorithm would produce as it traces out that parse path.

Definition 38. Let $\mathbf{P} = (U, J, s, D)$ be a parse path for the viable prefix β followed by a . Letting $U = (u_1, u_2, \dots, u_n)$, we let

$$u_k = [A_k \rightarrow \alpha_k \cdot \gamma_k, a] \text{ and } \pi_k = A_k \rightarrow \alpha_k \gamma_k$$

for $1 \leq k \leq n$. Letting $J = (j_1, j_2, \dots, j_m)$, we let:

$$j_k = [C_h \rightarrow \zeta_h \cdot D_h \eta_h] \text{ and } \phi_h = C_h \rightarrow \zeta_h D_h \eta_h,$$

for $1 \leq h \leq m$. Letting $D = (L_1, L_2, \dots, L_p)$, we let:

$$L_i = (\theta_i, k_i),$$

for $1 \leq i \leq p$. We first define six functions.

1. Define $V_\Pi: P \rightarrow \mathbb{Z}$ as

$$V_\Pi(\pi) = \begin{cases} |\alpha_k| + |\gamma_k| + 1 & \pi = \pi_k \text{ for some } k \\ -1 & \text{otherwise} \end{cases}.$$

2. Define $V_{\Pi, \epsilon}: P \rightarrow \mathbb{Z}$ as

$$V_{\Pi, \epsilon}(A \rightarrow \delta) = \begin{cases} |\delta| + 1 & A \rightarrow \delta \text{ is used in the derivation } \gamma_k \xrightarrow{*} \epsilon \text{ for some } k \\ -1 & \text{otherwise} \end{cases}.$$

3. Define $V_\Phi: P \rightarrow \mathbb{Z}$ as

$$V_\Phi(\phi) = \begin{cases} |\zeta_h| + 1 & \phi = \phi_h \text{ for some } h \\ -1 & \text{otherwise} \end{cases}.$$

4. Define $V_\Psi: P \rightarrow \mathbb{Z}$ as

$$V_\Psi(\psi) = \begin{cases} k_{h+1} & \psi = P(L_h, L_{h+1}) \text{ for some } h \\ -1 & \text{otherwise} \end{cases}.$$

5. Letting $\theta_h = X_{h,1}X_{h,2}\dots X_{h,q_h}$, define $V_{\Psi,\epsilon}: P \rightarrow \mathbb{Z}$ as

$$V_{\Psi,\epsilon}(E \rightarrow \kappa) = \begin{cases} |\kappa| + 1 & E \rightarrow \kappa \text{ is used in the derivation } X_{h,i} \xrightarrow{*} \epsilon \text{ for some } h \\ -1 & \text{otherwise} \end{cases}.$$

6. Letting $s = [D_s \rightarrow \gamma_s \cdot \delta_s]$, define $V_\Omega: P \rightarrow \mathbb{Z}$ as

$$V_\Omega(\omega) = \begin{cases} |\gamma_s| + |\theta_1| + 1 & \omega = D_s \rightarrow \gamma_s \delta_s \\ -1 & \text{otherwise} \end{cases}.$$

We have now come to our goal: define $L_{V,P}: P \rightarrow \mathbb{Z}$ as

$$L_{V,P}(\chi) = \max\{V_\Pi(\chi), V_{\Pi,\epsilon}(\chi), V_\Phi(\chi), V_\Psi(\chi), V_{\Psi,\epsilon}(\chi), V_\Omega(\chi)\}.$$

We call $L_{V,P}$ the **parse path conservation function**.

We intend V_P to correspond exactly to the function L_V , and vice versa. We can justifiably use the output of the Algorithm to determine the validity of a transformation if we can justifiably use the function L_V for that task. We will first recast the validity test for transformations in the next section, after which we will provide the promised justification.

4.2 An Alternative Test for Allowable Transformations

The conservation function of Section 2.4.2 may consider an infinite number of parse trees; thus, it is not self evident that any analysis of parse paths will be able to reproduce the conservation function, unless an infinite number are considered. In this section, we consider a subset of the parse trees considered by the conservation function which is finite in number and which does reproduce the conservation function. Moreover, this subset of parse trees will be “isomorphic,” in a sense, to the set of parse paths. After constructing this set of parse trees, and establishing the claimed properties, we will have shown that the parse path conservation function, and by extension, Algorithm 4, correctly calculate the conservation function in a finite amount of time and guarantee the successful execution of Algorithm 1.

Consider Definition 16, wherein we define the function $N_{a,T}$. In that section, we were given a TLR grammar and a sentential form, and we chose a sentence derivable from that sentential form. Proposition 1 justified our choice of an arbitrary sentence derivable from the given sentential form: the portion of the tree consisting of nodes that were descendants of nodes representing symbols in the original sentential form make no contribution to the value of the function $N_{a,T}$. By inspection of the definition of the function $N_{a,T}$, we can also conclude that nodes that are ordered greater than the node representing the symbol a make no such contribution either. Let us investigate what would happen to $N_{a,T}$ were we to remove those nodes from a parse tree.

Definition 39. Let $G = (\Sigma, N, P, S, T, M)$ be a TLR grammar and let $\alpha = \beta B a x$ be a sentential form for G such that $\beta \in (\Sigma \cup N)^*$ and $x \in \Sigma^*$, while B and a are a nonterminal and a terminal, respectively. Let y be a sentence in G such that $\alpha \xrightarrow{*} y$, and let T be the parse tree for y . Let A_1, A_2, \dots, A_n be the nodes representing the symbols βB , and let B and A represent the B and a , as they appear in α , respectively. We define two operators μ and μ' which acts on trees. The action of μ' is to remove all nodes X if either:

1. X is a descendent of some node Y , where $Y = A_i$ for some $1 \leq i \leq n$;
otherwise
2. X is not a descendent of any node Y , where $Y = A_i$ for some $1 \leq i \leq n$, and
in addition $X > A$.

Let μT be the tree formed from $\mu' T$ by replacing every leaf node labeled by a production $C \rightarrow \gamma$ with a node labeled C . The tree μT we shall refer to as the **simplified tree for βBa** . The operator μ is the **simple-tree projection operator**.

Definition 40. Let $G = (\Sigma, N, P, S)$ be a context-free grammar. Let T be a tree labeled with productions and grammar symbols from G , along with ϵ . Let N be an interior node in T that is labeled with the production $A \rightarrow \alpha$. If the child-string of N is a prefix of α , then we say that N is **parse-proper**.

Proposition 3. *Every simplified tree is parse-proper.*

Definition 41. Let T be a simplified tree for βBa . Let B and A be the nodes corresponding to B and a , respectively. Let U be the set of nodes that are ancestral to B . Let

$$\begin{aligned} Z = & \{C \text{ in } T : C \text{ corresponds to one of the symbols in } \beta B\} \\ & \cup \{C \text{ in } T : C \text{ is the least not autoancestral to } A \text{ but not } B\} \end{aligned}$$

For every $C \in Z$, the parent of C is in U . Let $W \subset U$ be such that, for every node $D \in W$, there exists some $C \in Z$ such that D is the parent of C . Let us put the elements of i into an ascending sequence we call the **prefix-ancestral sequence**: this sequence is (M_0, M_1, \dots, M_m) . For $0 \leq i \leq m$, define

$$V_i = \begin{cases} \{F \in U : F < M_1\} & i = 0 \\ \{F \in U : M_i < F < M_{i+1}\} & 1 \leq i < m \end{cases}.$$

Call $(V_i)_{i=0}^m$ the **prefix-ancestral interstitial sequence** for B and A . If, for all $0 \leq i \leq m$, there are no more than two distinct nodes A_1 and A_2 in V_i such that $\mathcal{L}(A_1) = \mathcal{L}(A_2)$, then we say that T is **proper above B** .

Definition 42. Let T be a simplified tree for βBa . Let B and A be the nodes corresponding to B and a , respectively. Let X be the set of nodes that are ancestral to A but not B . If there exist no more than two nodes $D_1, D_2 \in X$ such that $\mathcal{L}(D_1) = \mathcal{L}(D_2)$, then we say that T is **proper above A** .

Definition 43. Let T be a simplified tree for βBa . Let B and A be the nodes corresponding to B and a , respectively. If T is proper above both A and B , then we say that T is a **proper simplified tree for βBa** .

Definition 44. Let $G = (\Sigma, N, P, S, T, M)$ be a TLR grammar and let β be a viable prefix followed by the terminal a , for the nonterminal B is a nonterminal. Define a set of trees which we will refer to as the **simplified tree set for βa** as follows:

$$F_{\beta a} = \{T' : T' \text{ is a simplified tree for } \beta a\}.$$

We also define the following set of trees, which we will refer to as the **proper simplified tree set for βa** :

$$T_{\beta a} = \{T' : T' \text{ is a proper simplified tree for } \beta a\}.$$

Definition 45. Let $G = (\Sigma, N, P, S)$ be an LR(1) grammar. Let β be a viable prefix followed by a . Let $T \in T_{\beta a}$. Let $P \in T$, and let the children of P be A_1, A_2, \dots, A_n . Define

$$H_T(P) = \begin{cases} n+1 & P \text{ is an ancestor of } B \text{ but not } A \\ i & P \text{ is an ancestor of } A, \text{ and } A_i \text{ is autoancestral to } A \\ n+1 & P \text{ shares an ancestor with both } A \text{ and } B, \text{ and } B < P < A \end{cases}.$$

For $\pi \in P$ and $T \in F_{\beta a}$ define

$$M(T, \pi) = \max\{H_T(P) : \mathcal{L}(P) = \pi\}.$$

We define the **simplified conservation function for βa** ;

$$Y_{\beta a}(\pi) = \begin{cases} M(T, \pi) & \text{there exists some } T \in F_{\beta a} \text{ containing } P \text{ such that } \mathcal{L}(P) = \pi \\ -1 & \text{otherwise} \end{cases}.$$

Finally, we define the **proper simplified conservation function for βa** ;

$$Z_{\beta a}(\pi) = \begin{cases} M(T, \pi) & \text{there exists some } T \in T_{\beta a} \text{ containing } P \text{ such that } \mathcal{L}(P) = \pi \\ -1 & \text{otherwise} \end{cases}.$$

Definition 46. Let T be a tree, and let A , B , and C be three nodes such that A is ancestral to B , which is ancestral to C . A function which takes T , along with A , B , and C as argument, whose range is $\{0, 1\}$ we refer to as a **tree-projection decision function**.

Definition 47. Let T be a tree. We call f as a **triplet location function** if

$$f(T) = \begin{cases} (A, B, C) & , \\ \emptyset & \end{cases}$$

where A , B , and C are all nodes of T , such that A is ancestral to B , which is ancestral to C .

Since we give the nodes of a tree a total order, we can create a bijection between the nodes of a tree and the first n integers; thus, the return values of the triplet location function and the last three arguments to a tree-projection decision function are all elements of \mathbb{Z}_{n-1} .

Definition 48. Let $G = (\Sigma, N, P, S)$ be an LR(1) grammar, and let βB be a viable prefix followed by a . Let T be a parse-proper tree with yield βBa , letting B and A be the nodes corresponding to B and a , respectively. Let ρ be a tree-projection decision function, and let μ be a triplet location function. We define an operator $\Pi_{\rho, \mu}$ which will transform T . If $\mu(T) = \emptyset$, then $\Pi_{\rho, \mu}$ has no effect. Assume instead that $\mu(T) = (A, B, C)$; let the parents of A , B , and C be P_A , P_B , and P_C , respectively, should they all exist—in particular, P_A . The operation of $\Pi_{\rho, \mu}$ is as follows.

1. If $\rho(T, A, B, C) = 0$, then do one of the following:
 - (a) if A is the root of T , then make B the new root, but
 - (b) if A is not the root of T , then remove A as a child of P_A , and change the parent of B to P_A .

2. Otherwise, if $\rho(T, A, B, C) = 1$, then remove B as a child of P_B , and we change the parent of C to P_B .

We call $\Pi_{\rho, \mu}$ the **tree triplet surgery operator**.

If X is a totally ordered set, then we use the following total order on $X \times X \times \dots \times X \equiv X^n$. Let $(a_1, a_2, \dots, a_n), (b_1, b_2, \dots, b_n) \in X^n$; we say that $(a_1, a_2, \dots, a_n) \leq (b_1, b_2, \dots, b_n)$ in either of the following cases:

1. $a_i = b_i$ for $1 \leq i < j \leq n$, and $a_j < b_j$; or
2. $a_i = b_i$ for $1 \leq i \leq n$.

Definition 49. Let $G = (\Sigma, N, P, S)$ be an LR(1) grammar, and let βB be a viable prefix followed by a . Let T be a parse-proper tree with yield βBa , letting B and A be the nodes corresponding to B and a , respectively. Let $W = (M_0, M_1, \dots, M_m)$ be the prefix-ancestral sequence for B and A , and let $(V_i)_{i=0}^n$ be the prefix-ancestral interstitial sequence for B and A . We let $0 \leq j \leq m$ be the least index such that there are nodes $N_1, N_2, N_3 \in V_j$ such that

$$N_1 < N_2 < N_3, \text{ and} \\ \mathcal{L}(N_1) = \mathcal{L}(N_2) = \mathcal{L}(N_3).$$

We say that the three nodes (N_1, N_2, N_3) are a **repetitive triple for V_j** . If no such j exists, then let $\mu_B(T) = \emptyset$. If such a j does exist, let (N_1, N_2, N_3) be a repetitive triple for V_j , such that there does not exist a repetitive triple (M_1, M_2, M_3) satisfying

$$(M_1, M_2, M_3) < (N_1, N_2, N_3).$$

Call the repetitive triple (N_1, N_2, N_3) the **active repetitive triple** for T , and let $\mu_B(T) = (N_1, N_2, N_3)$.

Definition 50. Let $G = (\Sigma, N, P, S)$ be an LR(1) grammar, and let βB be a viable prefix followed by a . Let T be a parse-proper tree with yield βBa , letting B and A be the nodes corresponding to B and a , respectively. Let X be the set of nodes ancestral to A , but not ancestral to B . Let $C_1, C_2, C_3 \in X$ be three nodes such that

$$C_1 < C_2 < C_3, \text{ and} \\ \mathcal{L}(C_1) = \mathcal{L}(C_2) = \mathcal{L}(C_3),$$

Call (C_1, C_2, C_3) a **lookahead repetitive triple** (LA-repetitive triple) for T . If (C_1, C_2, C_3) is a LA-repetitive triple for T , such that there does not exist a lookahead repetitive triple (D_1, D_2, D_3) satisfying

$$(D_1, D_2, D_3) < (C_1, C_2, C_3),$$

then we refer to (C_1, C_2, C_3) as the **active LA-repetitive triple** for T . If there is no active LA-repetitive triple in T , then $\mu_A(T) = \emptyset$; if (C_1, C_2, C_3) is the active LA-repetitive triple, then let $\mu_A(T) = (C_1, C_2, C_3)$.

We will use one of only two constructions for tree-projection decision functions in the present work. Let ρ_0 be such that $\rho_0(T, A, B, C) = 0$ always. Our other construction for a tree-projection decision function is more complex. Let $G = (\Sigma, N, P, S)$ be an LR(1) grammar, and let βB be a viable prefix followed by a . Let T be a parse-proper tree with yield βBa , letting B and A be the nodes corresponding to B and a , respectively. Let $\pi \in P$, where $\pi = A \rightarrow \alpha$, and let $1 \leq n \leq |\alpha| + 1$. Let

C_1 , C_2 , and C_3 be three nodes such that C_1 is ancestral to C_2 , which is ancestral to C_3 . We define $\rho_{\pi,n}(T, C_1, C_2, C_3)$ presently: if there is a node N ancestral to C_2 , such that

- $C_1 < N < C_2$,
- $\mathcal{L}(N) = \pi$, and
- $H_T(N) = n$,

then let $\rho_{\pi,n}(T, C_1, C_2, C_3) = 0$; otherwise, let $\rho_{\pi,n}(T, C_1, C_2, C_3) = 1$.

Definition 51. Let $G = (\Sigma, N, P, S)$ be an LR(1) grammar, and let βB be a viable prefix followed by a . Let T be a parse-proper tree with yield βBa , letting B and A be the nodes corresponding to B and a , respectively. Let ρ be a tree-projection decision function. We now define two special tree triplet surgery operators; let $\Phi_\rho = \Pi_{\rho, \mu_B}$ and let $\Lambda_\rho = \Pi_{\rho, \mu_A}$. Let p and q be such that $\Phi_\rho^{p+1}T = \Phi^p T$ and $\Lambda_\rho^{q+1}T = \Lambda^q T$, respectively; refer to p and q as the **Φ -limit** and **Λ -limit** for T of Φ_ρ and Λ_ρ , respectively. Define $\Psi_\rho \equiv \Lambda_\rho^q \Phi_\rho^p$, an operator we refer to as the **proper projection operator**.

We will establish the following conventions. Let $G = (\Sigma, N, P, S)$ be an LR(1) grammar, and let βB be a viable prefix followed by a . If T is a parse-proper tree and P is a node in T , then we will use the symbol $\Psi_{T,P}$ to mean the operator Ψ_ρ , with $\rho = \rho_{\pi,n}$, where $\pi = \mathcal{L}(P)$ and $n = H_T(P)$. We use the symbol Ψ_0 to mean the operator Ψ_{ρ_0} .

Lemma 9. Let $G = (\Sigma, N, P, S)$ be an LR(1) grammar, and let βB be a viable prefix followed by a . If T is a parse-proper tree with yield βBa , then $\Phi_\rho^p T$ is proper above B , where p is the Φ -limit for T .

Proof. Let $G = (\Sigma, N, P, S)$ be an LR(1) grammar, and let βB be a viable prefix, followed by a , for $B \in N$. Let T be a parse-proper tree with yield βBa . Let ρ be a tree-projection decision function, and let p be the Φ -limit of Φ_ρ for T . Let $(V_i)_{i=0}^n$ be the prefix-ancestral interstitial sequence for B and A .

We proceed by induction on p . If $p = 0$, then there are no repetitive triples for T . There are thus no more than 2 distinct nodes B_1 and B_2 such that $\mathcal{L}(B_1) = \mathcal{L}(B_2)$. Therefore, T is proper above B . Since $\Phi_\rho T = T$, we conclude that $\Phi_\rho T$ is proper above B .

Assume that $\Phi_\rho^{p_Z} Z$ is proper above B for every tree Z with Φ -limit of p_Z , for some $p_Z \geq 0$. Assume also that $p = p_Z + 1$. Let (B_1, B_2, B_3) be the active repetitive triple for T . We replace one of these three nodes with one of the remaining two; since $\mathcal{L}(B_1) = \mathcal{L}(B_2) = \mathcal{L}(B_3)$, we conclude that $\Phi_\rho T$ is parse proper. As the Φ -limit of Φ_ρ for $\Phi_\rho T$ is $p - 1$, the induction hypothesis implies that $\Phi_\rho^{p-1} \Phi_\rho T$ is proper above B . ■

Lemma 10. Let $G = (\Sigma, N, P, S)$ be an LR(1) grammar, and let βB be a viable prefix followed by a . If T is a parse-proper tree with yield βBa , then $\Lambda_\rho^q T$ is proper above A , where q is the Λ -limit.

Proof. Let $G = (\Sigma, N, P, S)$ be an LR(1) grammar, and let βB be a viable prefix, followed by a , for $B \in N$. Let T be a parse-proper tree with yield βBa . Let ρ be a tree-projection decision function, and let q be the Λ -limit of Λ_ρ for T . Let X be the set of all nodes ancestral to A but not B .

We proceed by induction on q . If $q = 0$, then there are no LA-repetitive triples for T . Thus, there are at most two nodes D and D' in X such that $\mathcal{L}(D) = \mathcal{L}(D')$. Therefore, T is proper above A . Since $\Lambda_\rho T = T$, we conclude that $\Lambda_\rho T$ is proper above A .

Assume that, for any tree Y with a Λ -limit of Λ_ρ that is $q_Y \geq 0$, that we know that $\Lambda_\rho^{q_Y} Y$ is proper above A . Assume also that $q = q_Y + 1$. Let (C_1, C_2, C_3) be the active LA-repetitive triple for T . We know that $\mathcal{L}(C_1) = \mathcal{L}(C_2) = \mathcal{L}(C_3)$, and since whichever node is replaced gets replaced by one with the same label, we have a parse-proper tree in $\Lambda_\rho T$. The Λ -limit of Λ_ρ for $\Lambda_\rho T$ is $q - 1$. We therefore have, by the induction hypothesis, that $\Lambda_\rho^{q-1} \Lambda_\rho T$ is proper above A . \blacksquare

Let us consider an example. Let $G = (\Sigma, N, P, S)$ be the LR(1) grammar with

$$\begin{aligned}\Sigma &= \{q, r, h, j, c, b, k\}, \\ N &= \{S, H, G, A, B, D, Q\}, \text{ and} \\ P &= \{S \rightarrow H, \\ &\quad H \rightarrow QGr \mid Ak, \\ &\quad G \rightarrow Gh \mid GjH, \\ &\quad A \rightarrow cAD \mid B, \\ &\quad D \rightarrow \epsilon, \\ &\quad B \rightarrow b, \\ &\quad Q \rightarrow q\}.\end{aligned}$$

We now consider the string $x = qjqjqjccccbkrrhhhr$. It is easily verified that $x \in L(G)$. Let T be the parse tree for x . With the order that we have given trees in this work, we can represent T graphically as in Figure 2. We now consider a simplified tree and a proper simplified tree for the viable prefix $qjqjqjccccB$, when followed by k ; we have presented these trees in Figure 3.

We care about proper simplified trees because $Z_{\beta a}$ reproduces the conservation function, yet $T_{\beta a}$ is finite.

Theorem 6. *Let $G = (\Sigma, N, P, S, T, M)$. The proper simplified tree set is finite.*

Proof. Let $G = (\Sigma, N, P, S)$ be an LR(1) grammar, and let βB be a viable prefix followed by a , for $B \in N$. It will suffice to show that there is an upper bound to the height for elements of $T_{\beta Ba}$.

Let $T \in T_{\beta Ba}$. Let B and A be nodes in T corresponding to B and a , respectively. Let $(V_i)_{i=0}^n$ be the prefix-ancestral interstitial sequence for B and A .

Let $D \in N$ such that

$$D \xrightarrow{*} \epsilon, \tag{21}$$

where, for no $D' \in N$ with $D' \neq D$ is it the case that $D' \xrightarrow{*} \epsilon$ is a longer derivation than (21). Let N_D be the length of (21).

Let X be the greatest node ancestral to both B and A . Let B_a be the child of X autoancestral to B and let Q be those children of X greater than B_a . The greatest child of X is in Q ; let this child be A_Q .

Every element in $Q \setminus \{A_Q\}$ has a height not greater than N_D . Let Y_A be the subtree rooted at A_Q . Let X_A be the set of nodes in Y_A ancestral to A . By the condition that T is proper above A , we have that $|X_A| \leq 2|P|$. It is possible that the

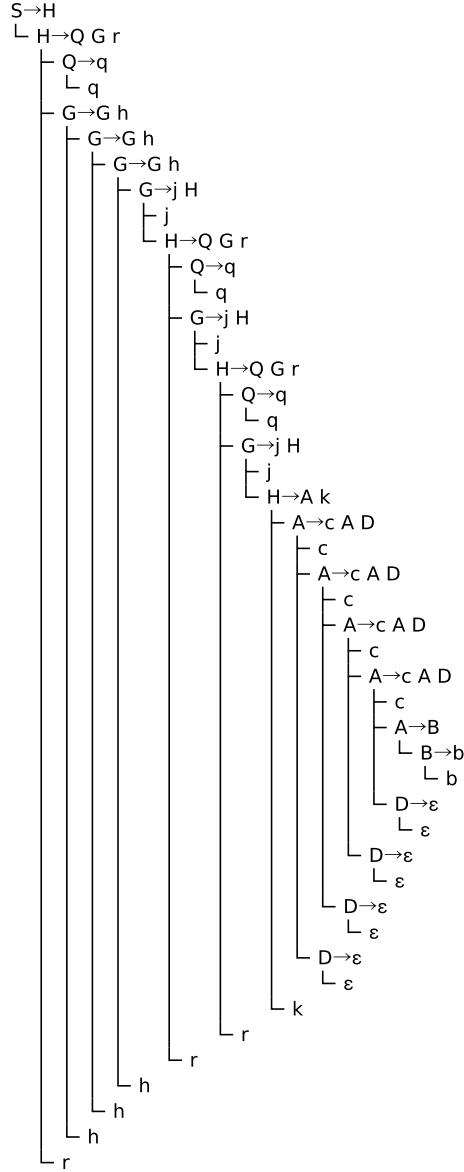


Figure 2: The parse tree for the string $qjqjqjcccbkrrhhhr$.

height of Y_A could exceed $|X_A| + 1$, by not by N_D . That is, the height of Y_A is less than $|X_A| + N_D$.

Let Z_B be the subtree with B_a as its root. Let γ be the yield of Z_B . Let $N_\gamma \subset N$ such that for all $C \in N_\gamma$, we have that $C \xrightarrow{*} \gamma$. Letting $N_\gamma = \{C_1, C_2, \dots, C_m\}$, let h_i be the height of the parse tree corresponding to the derivation $C_i \xrightarrow{*} \gamma$. Finally,

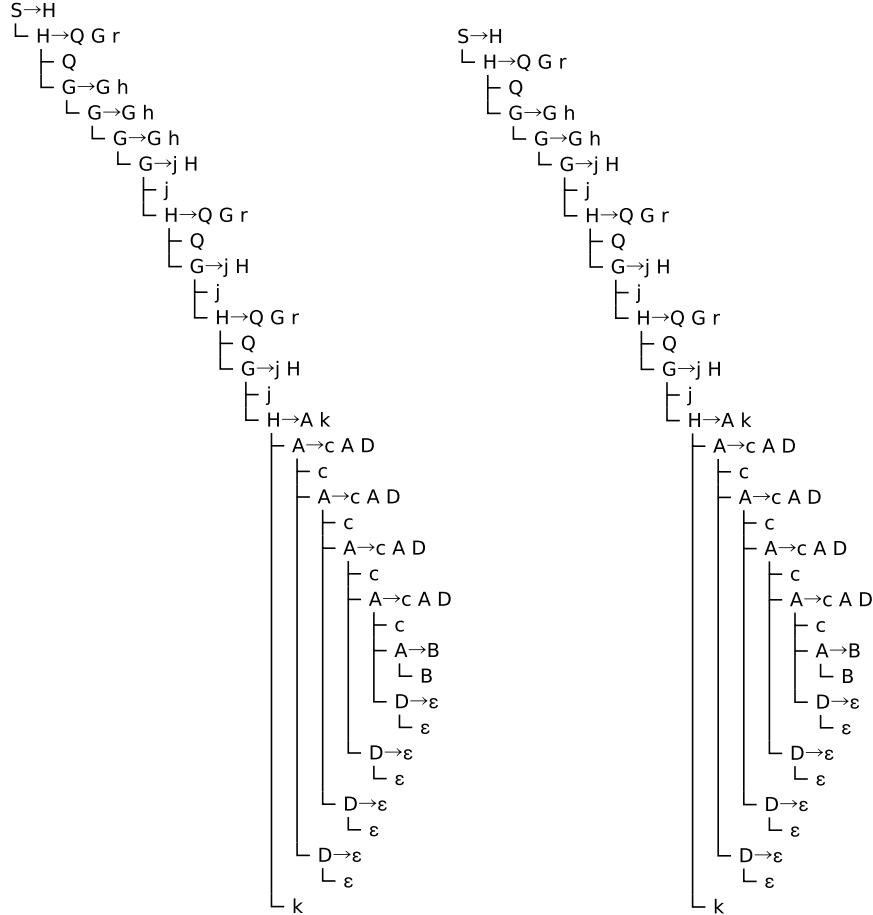


Figure 3: The simplified, and the proper simplified, trees for the viable prefix $qjqjqjccccB$, in former case, followed by the suffix $krrhhhr$, and in the later case, with lookahead k .

let $M_B = \max\{h_1, h_2, \dots, h_m\}$; clearly, the height of Z_B is less than M_B .

Let $\delta \in (\Sigma \cup N)^*$ such that $\delta\gamma = \beta B$. Within each V_i , for $0 \leq i \leq |\delta|$, there are not more than $2|P|$ nodes, by the condition that T is proper above B . Therefore, if we let T_S be the tree formed by removing from T all those nodes in Q , we can conclude that the height of T_S is not more than $2|P||\delta|$.

Combining these results, we see that the height of T is not greater than

$$2|P||\delta| + \max\{M_B, |X_A| + N_D\}.$$

Definition 52. Let $G = (\Sigma, N, P, S, T, M)$. Let P_0 be a production set over the terminal alphabet Σ and the nonterminal alphabet $N' \supset N$. If $\pi = A \rightarrow X_1 X_2 \dots X_m$ is a production in P such that $Y_{\beta\alpha}(\pi) \neq -1$, and either

- $\pi \in P_0$ if $Y_{\beta a}(\pi) = m + 1$; or

- there is some $\phi \in P_0$ such that $\phi = A \rightarrow X_1 X_2 \dots X_i S_1 S_2 \dots S_p$ where $i = Y_{\beta a}(\pi)$ if $Y_{\beta a}(\pi) < m + 1$;

then we say that P_0 **simply conserves βa for P** . If we replace $Y_{\beta a}$ with $Z_{\beta a}$, then we say that P_0 **simply conserves βa for P properly**.

Lemma 11. *Let the LR(1) grammar $G = (\Sigma, N, P, S)$, the viable prefix β ending in a nonterminal, and the terminal a , which follows β all be given. Let $T \in F_{\beta a}$. For any node P in T , such that $H_T(P) \neq -1$, there exists a tree $T' \in T_{\beta a}$ such that, for some node P' in T' , we have that*

$$H_T(P) = H_{T'}(P').$$

Proof. Let $\beta = \alpha B$, where B is a nonterminal. Let $T \in F_{\beta a}$, and let P in T , such that $H_T(P) \neq -1$. Let B and A correspond to B and a , respectively.

Let U be the set of nodes ancestral to B . Let $W = (M_0, M_1, \dots, M_m)$ be the prefix-ancestral sequence for B and A , and let $(V_i)_{i=0}^n$ be the prefix-ancestral interstitial sequence for B and A .

Let $\Omega \equiv \Psi_{T,P}$. It can easily be verified by inspecting the definition of $\Psi_{T,P}$ that there is a node P' in ΩT , corresponding to P , such that $H_{\Omega T}(P') = H_T(P)$.

The simplified tree ΩT is proper above A and it is proper above B (Lemmas 9 and 10). Since there is some node P_p in ΩT such that $H_{\Omega T}(P_p) = H_T(P)$, we have our conclusion. ■

Corollary 1. *Let the TLR grammar $G = (\Sigma, N, P, S, T, M)$, the viable prefix β , the terminal a , which follows β , and a production set P_0 all be given. Then P_0 simply conserves βa for P if and only if it does so properly.*

Theorem 7. *Let the TLR grammar $G = (\Sigma, N, P, S, T, M)$, the viable prefix β , the terminal a , which follows β , and the grammar transformation $\Delta \in \mathcal{A}_G$ be given. Let $\Delta G = (\Sigma, N_{\Delta G}, P_{\Delta G}, S, T, M)$. Then $\Delta \in \mathcal{V}_G(\beta a)$ if and only if $P_{\Delta G}$ simply conserves βa for P .*

Proof. Let $\Delta \in \mathcal{V}_G(\beta a)$. Let $x \in \Sigma^*$ be such that $\beta a x$ is a sentential form, and let $y \in \Sigma^*$ be such that $\beta a x \xrightarrow{*} y$. Let T be the parse tree for y , and let B and A be the nodes in T corresponding to the symbols B and a , respectively.

Let π be a free production; that is: $V_{\beta a}(\pi) = -1$. Thus, there are no nodes P in T meeting any of the first three criteria from Definition 16. Therefore, there are no simplified trees with a node P such that $\mathcal{L}(P) = \pi$.

Let $\pi = A \rightarrow \alpha$ be a conserved production that is not entirely conserved; thus, $V_{\beta a}(\pi) = n$, for $-1 \neq n \leq |\alpha|$. There is some node P ancestral to A such that $\mathcal{L}(P) = \pi$, where the n^{th} child of P is autoancestral to A ; this node does not get removed from T by μ (Definition 39). Therefore, in the tree μT , there is a node P_0 such that $H_{T_s}(P_0) = n$. We consider the possibility that $Y_{\beta a}(\pi) > n$.

Assume that $Y_{\beta a}(\pi) = n_0 > n$. That is, assume that there is some $T_1 \in F_{\beta a}$ —letting B_1 and A_1 be the nodes corresponding to the symbols B and a , respectively—such that there is a node F in T_1 that whose n_0^{th} child is autoancestral to A_1 . As β is a viable prefix followed by a , there exists some $x_1 \in \Sigma^*$ such that $\beta a x_1$ is a sentential form. Let $z \in \Sigma^*$ such that $\beta a x_1 \xrightarrow{*} z$; let T_z be the parse tree for z . There is an obvious injective mapping of nodes $q: T_1 \rightarrow T_z$ that preserves the structure of T_1 . The node $q(F)$ in T_z is such that $N_{\beta a x_1, T_z}(q(P)) = n_0$, a contradiction. Thus, $Y_{\beta a}(\pi) = n$.

Let $\phi = C \rightarrow \gamma$ be entirely conserved; since $\phi \in P_{\Delta G}$, we have that $P_{\Delta G}$ simply conserves βa for P . Therefore, the “If” direction is proved.

Let $P_{\Delta G}$ simply conserve βa for P . Let $\pi = A \rightarrow \alpha$ be such that

$$Y_{\beta a}(\pi) \leq |\alpha|.$$

Is it possible that $V_{\beta a}(\pi) > Y_{\beta a}(\pi)$? Assume one such x exists. Let $y \in \Sigma^*$ such that $\beta a x \xrightarrow{*} y$, and let T be the parse tree for y . We have assumed that there exists some node P in T such that

$$N_{\beta a x, T}(P) = Y_{\beta a}(\pi).$$

But, since $N_{\beta a x, T}(P) > -1$, it must be the case that either:

1. P is an ancestor of B ;
2. P is an ancestor of A ; or
3. P shares an ancestor with A and B , but is an ancestor of neither, such that $B < P < A$.

In any of these three cases, we would have that P would not be removed from T by μ . Hence, there is some $T_0 \in F_{\beta a}$ such that there exists P_0 in T_0 such that

$$H_{T_0}(P_0) > -1$$

and $\mathcal{L}(P) = \pi$, a contradiction.

Let π be such that $Y_{\beta a}(\pi) = n + 1$. Since $\pi \in P_{\Delta G}$, we have that $P_{\Delta G}$ conserves βa for P . \blacksquare

4.3 The Connection Between Parse Paths and Proper Simplified Trees

We are now ready to justify the method of Algorithm 4 as a means of determining if a grammar transformation is valid. We have modeled the operation of the Algorithm as an enumeration of parse paths, and we have examined a new formulation for determining if a transformation is valid. We now show that the method of Section 4.2 is just another way of looking at the operation of the Algorithm, in that each proper simplified tree corresponds to a parse tree, and visa versa.

4.3.1 A Mapping of Parse Paths to Proper Simplified Trees

Let G be a TLR grammar. We let \mathbb{T}_G be the set of all trees whose nodes are labeled either with a terminal, nonterminal or production from G . Let the set of all parse paths for the viable prefix β , followed by the terminal a be $\mathbb{P}_{\beta B, a, G}$. Additionally, let

$$\begin{aligned} \mathbb{U}_{\beta B, a, G} &= \bigcup_{(U, J, s, D) \in \mathbb{P}_{\beta B, a, G}} U, \\ \mathbb{J}_{\beta B, a, G} &= \bigcup_{(U, J, s, D) \in \mathbb{P}_{\beta B, a, G}} J, \\ \mathbb{S}_{\beta B, a, G} &= \bigcup_{(U, J, s, D) \in \mathbb{P}_{\beta B, a, G}} s, \text{ and} \\ \mathbb{D}_{\beta B, a, G} &= \bigcup_{(U, J, s, D) \in \mathbb{P}_{\beta B, a, G}} D. \end{aligned}$$

If G, β and a are understood, we omit them. If we are in a context where G, β and a are understood, then we use the symbols \mathbb{U}_0 and \mathbb{U}_1 as synonyms for \mathbb{J} and \mathbb{U} , respectively.

Definition 53. Let T , a nonempty tree in \mathbb{T} , be given. Let P be the greatest leaf in T ; if P is labeled with a grammar symbol, then let P_A be the parent of P ; if P is labeled with a production, then let $P_A = P$. Define P_A to be the **attach point for T** .

Definition 54. Let the TLR grammar $G = (\Sigma, N, P, S, T, M)$, and the viable prefix β , followed by the terminal a be given. Let $k \in \{0, 1\}$, and let $x \in \Sigma^*$ such that $|x| = k$. Define $Q_k: \mathbb{U}_k \rightarrow \mathbb{T}$ as follows: the value of $Q_k(U)$ —letting $U = (u_1, u_2, \dots, u_n)$, and letting $u_n = [A \rightarrow \alpha \cdot \gamma, x]$ —is given by one of the following 3 cases.

1. Assume that $|U| = 1$. In this case, we must have, by Rule 3 of Definition 29, that $\alpha = \epsilon$. Let $Q_k(U) = T_i$, where T_i is the one-node tree whose node is labeled $A \rightarrow \gamma$.
2. Assume that $|U| > 1$, and that $\alpha \neq \epsilon$. Let $\alpha = \alpha'X$ for some $X \in (\Sigma \cup N)$. In this case, we let $T = Q_k(u_1, u_2, \dots, u_{n-1})$, and we let P be the attach point for T . We let T' be that tree formed from T by adding a child labeled X to the children of P , such that this new node is greatest child of P in T ; let $Q_k(U) = T'$.
3. Assume that $|U| > 1$, and that $\alpha = \epsilon$. In this case, we let $V = Q_k(u_1, u_2, \dots, u_{n-1})$, and we let Q be the attach point for V . We let V' be that tree formed from V by adding a child labeled $A \rightarrow \gamma$ to the children of Q , such that this new node is greatest child of Q in V' ; let $Q_k(U) = V'$.

Call Q_k the **k -consumption function**.

Definition 55. Call the 1-consumption function the **upward link conversion function**; denote this function Q_U . Call the 0-consumption function the **ancestral link conversion function**; denote this function Q_A .

Definition 56. Let the TLR grammar G , and the viable prefix β , which is followed by a , be given. Let \mathbb{T}^* be the set of sequences of elements from \mathbb{T} . We will define $Q_D: \mathbb{D} \rightarrow \mathbb{T}^*$. Let $D \in \mathbb{D}$ be (L_1, L_2, \dots, L_n) . The value of $Q_D(D)$ is given by one of 2 cases.

1. Assume that $n = 1$. In this case, let $L_1 = (X_1 X_2 \dots X_p, k)$. For $1 \leq i < k$, let V_i be the parse tree for the derivation $X_i \xrightarrow{*} \epsilon$; let V_k be the single-node tree whose node is labeled X_k .
2. Assume that $n > 1$. In this case, let $L_1 = (X_1 X_2 \dots X_p, k)$. For $1 \leq i < k$, let V_i be the parse tree for the derivation $X_i \xrightarrow{*} \epsilon$. Let $Q_D(L_2, L_3, \dots, L_n) = (Y_1, Y_2, \dots, Y_h)$. Let V_k be the tree whose root:
 - is labeled $P(L_1, L_2)$, and
 - has Y_1, Y_2, \dots, Y_h as its children.

Define $Q_D(D) = (V_1, V_2, \dots, V_k)$. Call Q_D the **downward link conversion function**.

Definition 57. If G is a context-free grammar, and T is a parse-proper tree such that, for every interior node N , we have that $\mathcal{C}(N) = \mathcal{S}(N)$, then we say that T is **parse-complete**.

Proposition 4. *A parse tree is parse-complete.*

Definition 58. If $G = (\Sigma, N, P, S)$ is an LR(1) grammar, and T is a parse-proper tree then we define a function $F_U: \mathbb{T} \rightarrow \mathbb{T}$, where $F_U(T)$ is given by the following: for every interior node N such that $\mathcal{S}(N) \neq \mathcal{C}(N)$, we let $A_1, A_2, \dots, A_n \in N^*$ be such that $\mathcal{S}(N) = \mathcal{C}(N)A_1A_2 \dots A_n$, and we make the trees V_1, V_2, \dots, V_n be new children of N (in order), where for $1 \leq i \leq n$, the tree V_i is the parse tree for the derivation $A_i \xrightarrow{*} \epsilon$.

Proposition 5. *If U is an upward link, then $F_U(Q_U(U))$ is parse-complete.*

Definition 59. Let (U, J, s, D) be a parse path for the viable prefix βB followed by a . We will define $R: \mathbb{P} \rightarrow \mathbb{T}$. The value of $R(U, J, s, D)$ is as follows. Let $T_U = F_U(Q_U(U))$, let $Y = Q_D(D)$, and let $T_J = Q_A(J)$. There are then two cases to consider.

1. If we have that $|U| = 0$, then let T_0 be the tree formed by:
 - (a) attaching a node labeled B to the tree as the greatest child of the attach point for T_J ; and then
 - (b) attaching each tree from Y , in order, following the node added in 1a, to the attach point for T_J .
2. If we have that $|U| > 0$, then let T_0 be the tree formed by:
 - (a) attaching the root node of T_U to the tree as the greatest child of the attach point for T_J ; and then
 - (b) attaching the root node of each tree from Y , in order, following the node added in 2a, to the attach point for T_J .

Define $R(I, J, s, D) = T_0$; we call R the **parse-path conversion function**.

Lemma 12. *If (U, J, s, D) is a parse-path, then $R(U, J, s, D)$ is parse-proper.*

Proof. Let $k \in \{0, 1\}$, and let $x \in \Sigma^*$ such that $|x| = k$. Let $P_k \in \mathbb{I}_k$, where $P_k = (p_1, p_2, \dots, p_n)$. We proceed by induction on n . Since there are no interior nodes in $Q_k(P_k)$ when $|P_k| < 2$, we can use $|P_k| = 2$ as our basis step. The first two items of P_k are of the form

$$\begin{aligned} &[A_1 \rightarrow \cdot A_2 \alpha_1, x], \text{ and} \\ &[A_2 \rightarrow \cdot \alpha_2, x], \end{aligned}$$

respectively. Let $T_Q = Q_k(P_k)$. Since the only child of the root, which is the only interior node, is labeled $A_2 \rightarrow \alpha_2$, the child-string of the root is A_2 , which is a prefix of $A_2 \alpha_1$.

Let $P'_k \in \mathbb{I}_k$ be a parse-precession, such that $|P'_k| \geq 2$. Assume that we know that $Q_k(P_k)$ is a parse-proper tree; we now assume that $|P_x| = |P'_k| + 1$. Let T_P be the tree obtained after $|P'_k|$ applications of Q_k . There are two possibilities for p_n .

1. Assume that p_n is of the form $[A \rightarrow \alpha X \cdot \gamma, x]$. In this case, the attach point P_A will be labeled $A \rightarrow \alpha X \gamma$. The child-string of P_A will be α . The final application of Q_k will attach a node labeled X as the greatest child of P_A , forming the tree T_0 . The parent of this new node—the counterpart of P_A in T_0 —has the child-string αX , which is a prefix of $\alpha X \gamma$.

2. Assume that p_n is of the form $[C \rightarrow \cdot\delta, x]$. In this case, the attach point of T_P will be labeled $[D_A \rightarrow \zeta_A \cdot C\eta_A, x]$. The child-string of P_A is ζ_A . After the final application of Q_k , the child-string will be $\zeta_A C$, which is a prefix of $\zeta_A C\eta_A$, as required.

Thus, we have established that $Q_U(U)$ and $Q_A(J)$ are parse-proper trees.

We turn now to the downward link. Let $D = (L_1, L_2, \dots, L_p)$. Assume that $p = 1$. In this case, the value of Q_D is a sequence of trees $Y = (T_1, T_2, \dots, T_m)$. The trees T_1, T_2, \dots, T_{m-1} are all parse trees, so they are all parse-proper. In this case, the tree T_m will be a single-node tree, hence, it is trivially parse-proper.

Assume now that all elements of $Q_D(D')$ are parse-proper if $|D'| \geq 1$. Assume also that $p = |D'| + 1$. After $p - 1$ applications of Q_D , we have a downward link (θ, k_θ) and a sequence $Y = (V_1, V_2, \dots, V_q)$. By the induction hypothesis, each element of Y is a parse-proper tree. The penultimate application of Q_D will result in a sequence of k_θ trees: the first $k_\theta - 1$ trees are parse-proper, as they are parse trees. We know that $P(L_1, L_2)$ is a production, where

$$P(L_1, L_2) = A \rightarrow \mathcal{L}(\mathcal{R}(V_1))\mathcal{L}(\mathcal{R}(V_2)) \dots \mathcal{L}(\mathcal{R}(V_{k_\theta}))X_1X_2 \dots X_{n_\theta}.$$

The next application of Q_D will result in a sequence of trees, all but the last of which are clearly parse trees; let the last tree in this sequence be V' . Since the elements of Y are added in order, we have, letting the children of the root of V' be labeled $C_1, C_2, \dots, C_{k_\theta}$, we can see that

$$\begin{aligned} \mathcal{L}(C_1) &= \mathcal{L}(\mathcal{R}(V_1)), \\ \mathcal{L}(C_2) &= \mathcal{L}(\mathcal{R}(V_2)), \\ &\text{etc.} \end{aligned}$$

Since this is a prefix of the right side of $P(L_1, L_2)$, we have shown that every element of $Q_D(D)$ is parse-proper.

There are two ways that three trees $Q_A(J)$ and $Q_U(U)$ are combined with the trees in $Q_D(D)$ to form the tree $R(U, J, s, D)$. Let $s = [A_\sigma \rightarrow \alpha_\sigma X_\sigma \cdot \gamma_\sigma]$.

1. Assume that $|U| = 0$. In this case, we know that $X_\sigma = B$. By Rule 1a of Definition 59, we add a node labeled B to the attach point.
2. Assume instead that $|U| > 0$. In this case, we know that $u_1 = [X_\sigma \rightarrow \cdot\kappa_1, a]$. Thus, $\mathcal{R}(Q_U(U)) = X_\sigma$. By Rule 2a of Definition 59, we add a tree—namely, the tree $Q_U(U)$ —whose root is labeled $X_\sigma \rightarrow \kappa_1$.

The attach point $P_{0,A}$ of $Q_A(J)$ is labeled $A_\sigma \rightarrow \alpha_\sigma X_\sigma \gamma_\sigma$; since $Q_A(J)$ is parse-proper, the child-string of the attach point must be α_σ . So after the application of either Rule 1a or Rule 2a, whichever is appropriate, we have shown that the child string $P_{0,A}$ now has the child-string $\alpha_\sigma X_\sigma$.

Now, L_1 is a link, either partial-downward or terminal, for γ_σ . We can write

$$\gamma_\sigma = X_1X_2 \dots X_{k_\theta}X_{k_\theta+1} \dots X_{n_\theta}.$$

After the application of Rule 1a or Rule 2a, as appropriate, we add the elements of

$Q_D(D)$ to the attach point. Clearly,

$$\begin{aligned} X_1 &= \mathcal{L}(\mathcal{R}(V_1)), \\ X_2 &= \mathcal{L}(\mathcal{R}(V_2)), \\ &\vdots \\ X_{k_\theta-1} &= \mathcal{L}(\mathcal{R}(V_{k_\theta-1})); \end{aligned}$$

but what of X_{k_θ} and V_{k_θ} ? If L_1 is a terminal link, then V_{k_θ} is a one-node tree whose node is labeled $X_{k_\theta} = a$. Otherwise, the root of V_{k_θ} is labeled $P(L_1, L_2)$, which is $X \rightarrow \lambda$. In either case, $\mathcal{L}(\mathcal{R}(V_{k_\theta})) = X_{k_\theta}$. ■

Definition 60. Let the TLR grammar G , and the parse-proper tree T be given. Let $L = (K_1, K_2, \dots, K_n)$ be the set of leaves, such that $K_1 < K_2 < \dots < K_n$. If $\mathcal{L}(K_i) \in \Sigma \cup N \cup \{\epsilon\}$ for all $1 \leq i \leq n$, then we let

$$\alpha = \mathcal{L}(K_1)\mathcal{L}(K_2)\dots\mathcal{L}(K_n).$$

We call α the **yield** of T .

Lemma 13. The yield of $R(U, J, s, D)$ is βBa .

Proof. Let $Q_D(D) = (V_1, V_2, \dots, V_{n_d})$. The yield of any of the trees $V_1, V_2, \dots, V_{n_d-1}$ is clearly ϵ . The yield of V_{n_d} is clearly a .

Let $k \in \{0, 1\}$, and let $x \in \Sigma^*$ such that $|x| = k$.

Let $P_k = (p_1, p_2, \dots, p_n)$ be a k -parse precession. We note that the only way a grammar symbol leaf is added to the tree $T = Q_k(P_k)$ is when we evaluate Q_k on an item of the form

$$[A \rightarrow \alpha X \cdot \beta, x] \tag{22}$$

If $p_{i_1}, p_{i_2}, \dots, p_{i_m}$ is the set of all items of the form (22), such that $i_1 < i_2 < \dots < i_m$, where

$$p_{i_h} = [A_h \rightarrow \alpha_h X_h \cdot \beta_h, x],$$

for $1 \leq h \leq m$, then the yield of $Q_k(P_k)$ is evidently the string $X_1 X_2 \dots X_m$. But this is just the trace of P_k . Therefore, the concatenation of the yield of $Q_A(J)$ and $Q_U(U)$ is βB , by Theorem 5.

The yield of $R(U, J, s, D)$ is clearly the yield of $Q_A(J)$, $Q_U(U)$, and V_{n_d} , concatenated, but this is just βBa . ■

Lemma 14. If $R(U, J, s, D)$ is a parse path, then $R(U, J, s, D)$ is a simplified tree.

Proof. We wish to find a sentence $y \in L(G)$ such that, letting T be the parse tree for y , we have that $\Psi_0 T = R(U, J, s, D)$.

Let \mathcal{N} be the set of all leaves of $\Psi_0 T$ that are labeled with a nonterminal. For any nonterminal A , let F_A be the parse tree for the derivation

$$A \xrightarrow{*} z_A,$$

for some $z_A \in \Sigma^*$. Let T_1 be that tree formed from T by replacing every node $N \in \mathcal{N}$ with the tree $F_{\mathcal{L}(N)}$.

Let \mathcal{D} be the set of all production nodes J_0 in T_1 such that the child-string of $\mathcal{C}(J_0) \neq \mathcal{S}(J_0)$. For any such node J_0 , let $\mathcal{S}(J_0) = \alpha_0 Z_1 Z_2 \dots Z_m$, where

$\alpha_0 = \mathcal{C}(J_0)$; let Z_{J_0} be the sequence of trees $(F_{Z_1}, F_{Z_2}, \dots, F_{Z_m})$. Let T_2 be the tree formed by, for each $J_f \in D$, adding the elements of Z_{J_f} as children of J_f , in order.

Let z be the yield of T_2 . Clearly, the parse tree of z is T_2 . Recalling the simple-tree projection operator μ , consider the tree μT_2 . We will examine the changes to T_2 made by μ .

For any node N_z such that N_z corresponds to one of the symbols in β , we note that N_z would be removed and replaced with a node labeled $\mathcal{L}(N_z)$ by μ . We would also remove any node M_z that is not an ancestor of any node representing one of the symbols in βa , provided that $M_z > A$, where A is the node representing a .

We consider the conditions under which we will add nodes to T_1 in the construction of T_2 . Let H_x be a node in T_1 , such that we add children to it when constructing T_2 . There are two cases:

1. Assume that H_z is some node in $Q_A(J)$. In this case, we note that in $Q_A(J)$ —letting H_j be the node corresponding to H_z in $Q_A(J)$ —the greatest child of H_j is an ancestor of the greatest tree in $Q_D(D)$. Let this child be the i^{th} child of H_j in. Clearly, the simplified tree must retain the first i children.
2. Assume that H_z corresponds to some node in one of the trees in $Q_D(D)$. We must have that H_z corresponds to a node in the final element of $Q_D(D)$. Let this node be H_D . Let A_D be the node in the final element of $Q_D(D)$ that is labeled a . We can easily show the following: “if $\mathcal{S}(H_D) \neq \mathcal{C}(H_D)$, then the greatest child of H_D is autoancestral to A_D .” Thus, any nodes added as children of H_D during the construction of T_2 will be removed when μ is applied to T_2 .

Can any node be removed from the part of T_2 corresponding to $Q_U(U)$? By Proposition 5, we have that there are no production nodes in $F_U(Q_U(U))$ that would have children removed.

By Lemma 13, we can say that the only nodes which will be replaced during the construction of T_1 from T are the nodes corresponding to symbols in β which are labeled with nonterminals. Let N_β be one of the nonterminal nodes that is replaced by a production nodes. The head of the production will be $\mathcal{L}(N_\beta)$, and when we apply μ to T_2 , we will this replacement node with a node labeled $\mathcal{L}(N_\beta)$: that is, we will revert to N_β .

So, we have examined all nodes and changes made applying μ to T_2 , and we have arrived back at T . Therefore, $R(U, J, s, D)$ is the simplified tree for βa . ■

Theorem 8. *If (U, J, s, D) is a parse path, then $R(U, J, s, D)$ is a proper simplified tree.*

Proof. Let $G = (\Sigma, N, P, S)$ be an LR(1) grammar, and let βB be a viable prefix, followed by a , where $B \in N$. Let $(U, J, s, D) \in \mathbb{P}_{\beta B, a, G}$, and let $T = R(U, J, s, D)$. Let A and B be the nodes in T corresponding to the symbols B and a .

There are two ways that T may fail to be proper: it may be improper above either A or B .

Assume the former, for the sake of a contradiction. Let $D = (L_1, L_2, \dots, L_n)$. This means that there are three distinct nodes $P_{A,1}$, $P_{A,2}$, and $P_{A,3}$ in T , each of which are ancestral to A but not B , such that $\mathcal{L}(P_{A,1}) = \mathcal{L}(P_{A,2}) = \mathcal{L}(P_{A,3})$. Assume, without loss of generality, that $P_{A,1} < P_{A,2} < P_{A,3}$. There are thus three partial downward links L_{i_1} , L_{i_2} , and L_{i_3} , where $i_1 < i_2 < i_3$, corresponding to $P_{A,1}$,

$P_{A,2}$, and $P_{A,3}$, respectively. Now,

$$\begin{aligned}\mathcal{L}(P_{A,1}) &= P(L_{i_1}, L_{i_1+1}), \\ \mathcal{L}(P_{A,2}) &= P(L_{i_2}, L_{i_2+1}), \text{ and} \\ \mathcal{L}(P_{A,3}) &= P(L_{i_3}, L_{i_3+1}).\end{aligned}$$

However, the fact that $P(L_{i_1}, L_{i_1+1}) = P(L_{i_2}, L_{i_2+1}) = P(L_{i_3}, L_{i_3+1})$ is in contradiction with Definition 36. Therefore, T is proper above A .

We now consider the latter of the two ways in which T may fail to be proper: that is, assume that T is not proper above B .

We know, from the previous two Lemmas, that T is a simplified tree for βBa . Let W and $(V_i)_{i=0}^q$ be the prefix-ancestral set and prefix-ancestral interstitial sequence, respectively.

We have assumed that there is some i such that, within V_i , there are three nodes C_1, C_2 , and C_3 such that

$$\begin{aligned}C_1 &< C_2 < C_3, \text{ and} \\ \mathcal{L}(C_1) &= \mathcal{L}(C_2) = \mathcal{L}(C_3).\end{aligned}$$

Depending on whether or not the nodes in V_i are ancestral to A or not, we define k , x , and I as follows:

- if the nodes in V_i are ancestral to A , then let $k = 0$, $x = \epsilon$, and $I = J$;
- otherwise, the nodes in V_i are not ancestral to A , in which case we let $k = 1$, $x = a$, and $I = U$.

Either way, let $I = (r_1, r_2, \dots, r_m)$.

Note that C_1 has no child labeled with a grammar symbol; thus it was created when Q_k was applied to an item of the form $[A_I \rightarrow \cdot \alpha_I, x]$; likewise for C_2 and C_3 . The three nodes C_1, C_2 , and C_3 thus correspond to three items r_{t_1}, r_{t_2} , and r_{t_3} , respectively. When $t_1 < q_t < t_3$, we have that $r_{q_t} \triangleright_k r_{q_t+1}$, as required by Condition 4 of Definition 29, yet we have that $r_{t_1} = r_{q_t} = r_{t_2}$ when $q_t = t_2$, in violation of that Condition.

Therefore, we have shown that T is proper above B because the argument of the last paragraph applies to $Q_U(U)$ and $Q_A(J)$. Since T is proper above A , it is in fact a proper simplified tree. ■

4.3.2 The Interchangeability of Parse Trees and Parse Paths

Theorem 9. *There is a bijection between the set of all parse paths and the proper simplified tree set for βa .*

Proof. Let $G = (\Sigma, N, P, S, T, M)$ be a TLR grammar, let $\beta \in (\Sigma \cup N)^*$ and $B \in N$ be such that βB is a viable prefix followed by a .

The bijection is R . We first begin by proving that R is surjective.

Let T_s be a proper simplified tree for βB followed by a . We let A and B be the nodes corresponding to a and B , respectively. Let S be the greatest node in T_s such that S is an ancestor of both B and A . Let R_B be that child of S that is autoancestral to B . Let T_J be that tree formed by removing R_B from T_s , along with the other children of S that are greater than R_B . Let T'_U be the subtree rooted at

R_B . Finally, letting V_1, V_2, \dots, V_{n_v} be those children of S greater than R_B we let $Y = (V_1, V_2, \dots, V_{n_v})$.

Let T_U be that tree formed from T'_U by removing all nodes N_ϵ if N_ϵ is the root of a subtree with ϵ -yield.

Let α_D be the right side of $\mathcal{L}(S)$. We define a function $Q_D^{-1} : (\mathbb{T}^* \times (\Sigma \cup N)^*) \rightarrow \mathbb{D}$ now. The value of $Q_D^{-1}(Y_D, \alpha_D)$ we give now, letting $Y_D = (V_{D,1}, V_{D,2}, \dots, V_{D,n_D})$.

1. If V_{D,n_D} has but a single node, then let L be a terminal link from α_D to a , with the value (α_D, n_D) . In this case, define $Q_D^{-1}(Y_D, \alpha_D) = (L)$.
2. If V_{D,n_D} has multiple nodes, then we first let L' be a partial downward link from α_D to a , with the value (α_D, n_D) . Next, we label the children of the root of V_{D,n_D} as follows (in order): W_1, W_2, \dots, W_{m_D} , and we label the right side of $\mathcal{L}(\mathcal{R}(V_{D,n_D}))$ as α' . Let $(L_1, L_2, \dots, L_{p_D}) = Q_D^{-1}((W_1, W_2, \dots, W_{m_D}), \alpha')$. Define $Q_D^{-1}(Y_D, \alpha_D) = (L', L_1, L_2, \dots, L_{p_D})$.

We will pause and establish an intermediate result. Let $Y \in \mathbb{T}^*$ be such that, should we let $Y = (V_1, V_2, \dots, V_{m_l})$, each of $V_1, V_2, \dots, V_{m_l-1}$ is a parse tree with ϵ -yield and V_{m_l} is a parse-proper tree whose yield is a ; furthermore,

$$\mathcal{L}(\mathcal{R}(V_1)) \mathcal{L}(\mathcal{R}(V_2)) \dots \mathcal{L}(\mathcal{R}(V_{m_l}))$$

is a prefix of α . We will show, by induction on the height of V_{m_l} , that

$$Q_D(Q_D^{-1}(Y, \alpha)) = Y. \quad (23)$$

Let the height of $V_{m_l} \equiv h_V$. Assume that $h_V = 1$. In this case, V_{m_l} has only a single node, and that is labeled a . Thus, the value of $Q_D^{-1}(Y, \alpha)$ is the terminal link (α, m_l) . We have, by Rule 1 of Definition 56, that $Q_D(Q_D^{-1}(Y, \alpha)) = Y'$, where we are letting $Y' = (Q_1, Q_2, \dots, Q_{m'_l})$. However, we note that $|Q_{m'_l}| = m_l$, and that the first $m_l - 1$ elements of Y' are parse trees for the derivations

$$\begin{aligned} \mathcal{L}(\mathcal{R}(V_1)) &\xrightarrow{*} \epsilon, \\ \mathcal{L}(\mathcal{R}(V_2)) &\xrightarrow{*} \epsilon, \\ &\vdots \\ \mathcal{L}(\mathcal{R}(V_{m_l-1})) &\xrightarrow{*} \epsilon, \end{aligned}$$

respectively. Finally, since V_{m_l} is the one-node tree whose root is labeled a —again, by Rule 1 of Definition 56—we have (23) if $h_V = 1$.

Assume now that we have established (23) if $h_V = k_V$, where $k_V \geq 1$; assume also that $h_V = k_V + 1$. When considering the evaluation of Q_D^{-1} on Y , we recursively evaluate of Q_D^{-1} on (Y_h, α_h) , such that the final element of Y_h is of height k_V . We can therefore say, by the induction hypothesis, that $Q_D(Q_D^{-1}(Y_h, \alpha_h)) = Y_h$. We return to the evaluation of Q_D . Note that L_1 is a partial downward link for α to a , the value of which we denote (α_1, m_l) . We denote $\alpha = X_1 X_2 \dots X_{|\alpha|}$. Let $V'_1, V'_2, \dots, V'_{m_l-1}$ be parse trees for the derivations

$$\begin{aligned} X_1 &\xrightarrow{*} \epsilon, \\ X_2 &\xrightarrow{*} \epsilon, \\ &\vdots \\ X_{m_l-1} &\xrightarrow{*} \epsilon, \end{aligned}$$

and let V'_{m_l} be the tree formed by making each of the elements of Y_h children of a node labeled $X_{m_l} \rightarrow \gamma$. However, this is just Y , so we have (23).

Let $k \in \{0, 1\}$, and let $y \in \Sigma^*$ such that $|y| = k$. Let $\mathbb{T}_P \subset \mathbb{T}$ be the set of all parse-proper trees whose leaves are either a grammar symbol or ϵ . We define $Q_{k,y}^{-1}: \mathbb{T}_P \rightarrow \mathbb{U}_k$ for some $T_x \in \mathbb{T}_P$ as follows.

1. Assuming that T_x has no nodes, we let

$$Q_{k,y}^{-1}(T_x) = ().$$

2. Assume that T_x has multiple nodes, and assume also that the greatest leaf is labeled by the production $A_x \rightarrow \alpha_x$. Let T'_x be that tree formed from T_x by removing the latter's greatest leaf, and let $(i_1, i_2, \dots, i_{n_r}) = Q_{k,y}^{-1}(T'_x)$. Finally, let

$$Q_{k,y}^{-1}(T_x) \equiv (i_1, i_2, \dots, i_{n_r}, [A_x \rightarrow \cdot \alpha_x, y])$$

3. Assume that T_x has multiple nodes, and assume also that the greatest leaf N_x is labeled by the grammar symbol X_x ; let the parent of N_x be labeled $E \rightarrow \gamma_x$. Let $m_x - 1$ be the number of siblings of N_x , and let T''_x be that tree formed from T_x by removing N_x . Let

$$(j_1, j_2, \dots, j_{p_x}) = Q_{k,y}^{-1}(T''_x).$$

Let $\gamma_x = Z_1 Z_2 \dots Z_{s_x}$, and let

$$j' = [E \rightarrow Z_1 Z_2 \dots Z_{m_x} \cdot Z_{m_x+1} \dots Z_{s_x}, y].$$

Finally, let

$$Q_{k,y}^{-1}(T_x) \equiv (j_1, j_2, \dots, j_{s_x}, j').$$

We establish an intermediate result. Let $k \in \{0, 1\}$, and let $z \in \Sigma^*$ such that $|z| = k$. Let T_y be a parse-proper tree, and whose root is a production node. We wish to show that

$$Q_k(Q_{k,z}^{-1}(T_y)) = T_y \tag{24}$$

Letting the number of nodes in T_y be n_y , we proceed by induction on n_y . Assume, for our basis step, that $n_y = 1$. In this case, the only node is labeled $A_y \rightarrow \alpha_y$. The application of $Q_{k,z}^{-1}$ yields the sequence

$$([A_y \rightarrow \cdot \alpha_y, z]);$$

this is the input of Q_k . The application of Q_k , as given by Rule 1 of Definition 54, will create a one-node tree whose node is labeled $A_y \rightarrow \alpha_y$. This is just T_y .

Assume that we know that (24) holds for k_y -node trees, where $k_y \geq 1$. Assume also that $n_y = k_y + 1$. We consider the greatest leaf of T_y , a leaf that we label F_y . This leaf may be either a grammar symbol or a production node—we consider these cases separately.

1. Assume that $\mathcal{L}(F_y) = C_y \rightarrow \gamma_y$. Let T'_y be that tree formed from T_y by removing F_y ; in order to evaluate $Q_{k,z}^{-1}$ on T_y , we first evaluate $Q_{k,z}$ on T'_y . By the induction hypothesis, $Q_k(Q_{k,z}^{-1}(T'_y)) = T'_y$. The value of $Q_{k,z}^{-1}(T_y)$ will be the item $[C_y \rightarrow \gamma_y, z]$ appended to the sequence $Q_{k,z}^{-1}(T'_y)$. Consider the evaluation of Q_k when we evaluate the expression $Q_k(Q_{k,z}^{-1}(T_y))$: we first recursively evaluate of Q_k with the sequence of items $Q_{k,z}^{-1}(T'_y)$, which will yield T'_y ; the evaluation of Q_k will be completed by adding a node, corresponding to the item $[C_y \rightarrow \cdot \gamma_y, z]$ to T'_y ; this node will be labeled $C_y \rightarrow \gamma_y$, hence, $Q_k(Q_{k,z}^{-1}(T_y)) = T_y$.

2. Assume that $\mathcal{L}(F_y) = X$, for some $X \in \Sigma \cup N$. This case is similar to the first. The label of the parent of F_y we give the label $D_y \rightarrow \delta_y X \zeta_y$, such that F_y has $|\delta_y|$ siblings. Let T'_y be that tree formed from T_y be removing F_y . As before, when evaluating $Q_k(Q_k^{-1}(T_y))$, we form the final tree from T'_y and the item $[D_y \rightarrow \delta_y X \cdot \zeta_y, z]$; this yields T_y .

We can appeal to this result directly to conclude that there is a function Q_A^{-1} such that, for any parse-proper tree T_a , we have that

$$Q_A(Q_A^{-1}(T_a)) = T_a. \quad (25)$$

Similarly, there is a function Q_u^{-1} such that

$$Q_U(Q_U^{-1}(T_u)) = T_u.$$

We now return to the trees we considered in the beginning of this proof. Let $D = Q_D^{-1}(Y)$, and let $J = Q_A^{-1}(T_J)$. Let $\pi = \mathcal{L}(S)$, recalling that S is a node in the tree T_s . Let the index of the child of S that is autoancestral to B be i_b , we write $\pi = F \rightarrow \eta\theta$, such that $|\eta| = i_b$; thus we let $s = [F \rightarrow \eta \cdot \theta]$. Finally, we turn to U : if T_U has a single node, then we let $U \equiv ()$, otherwise, T_U has multiple nodes, in which case we let $U \equiv Q_U^{-1}(T_U)$.

Let $V \equiv R(U, J, s, D)$. If we can show that $V_s = T_s$, then we shall have shown that R is surjective. We first note that $T_J = Q_A(J)$ and $Y = Q_D(D)$. Now there are two cases to consider.

1. Assume that T_U has but a single node. In this case, we attach a single node labeled B as the greatest child to T_J , as in Rule 1 of Definition 59; but since T_U has but a single node, then this node must be labeled B .
2. Assume that T_U has multiple nodes. If this is the case, then $T_U = Q_U(U)$. We note that the tree $T_{U,0}$, as specified in Definition 59, is in fact equal to T'_U , as specified in this proof. In that Definition, we construct $R(U, J, s, D)$ according to Rule 2 by attaching T'_U to T_J at the place that it originally resided.

In either case, we finish by attaching the nodes from Y following the root of T'_U . Clearly, this yields T_s .

Assume now, for the sake of a contradiction, that R is not injective. That is: assume that there are two distinct parse-paths (U_1, J_1, s_1, D_1) and (U_2, J_2, s_2, D_2) such that

$$R(U_1, J_1, s_1, D_1) = R(U_2, J_2, s_2, D_2). \quad (26)$$

There are several ways in which these parse paths could differ. We examine each of these ways, eliminating each in turn.

Let P_k and R_k be two k -parse paths, such that $Q_k(P_k) = Q_k(R_k)$; let $w \in \Sigma^*$ such that $|w| = k$. We will show that $P_k = R_k$. Assume, for the sake of a contradiction, that $P_k \neq R_k$. Since the number of nodes in $Q_k(P_k)$ is $|P_k|$, and since $Q_k(P_k) = Q_k(R_k)$, we have that $|P_k| = |R_k|$. Let $P_k = (p_1, p_2, \dots, p_{n_p})$ and let $R_k = (r_1, r_2, \dots, r_{n_p})$. Let i_z be the least index such that $p_{i_z} \neq r_{i_z}$; let $p_h = [A_{P,h} \rightarrow \alpha_{P,h} \cdot \gamma_{P,h}, w]$ and let $r_h = [A_{R,h} \rightarrow \alpha_{R,h} \cdot \gamma_{R,h}, w]$, for $1 \leq h \leq n_R$. There are 2 cases to consider.

1. Assume that $i_z = 1$; thus, $\alpha_P = \alpha_R = \epsilon$. Since Q_k yields identical one-node trees when evaluated on the sequences (p_1) and (r_1) , we must have that $A_{P,1} \rightarrow \gamma_{P,1} = A_{R,1} \rightarrow \gamma_{R,1}$.

2. Assume that $i_z > 1$. As $p_{i_z-1} = r_{i_z-1}$, we have that

$$\begin{aligned} A_{P,i_z-1} &= A_{R,i_z-1}, \\ \alpha_{P,i_z-1} &= \alpha_{R,i_z-1}, \text{ and} \\ \gamma_{P,i_z-1} &= \gamma_{R,i_z-1}. \end{aligned}$$

We have either that:

(a) if $\alpha_{P,i_z} = \epsilon$, then the node corresponding to p_{i_z} is the first child of a node corresponding to p_{i_z-1} ; let N_P in $Q_k(P_k)$ and N_R in $Q_k(R_k)$ be the nodes corresponding to p_{i_z-1} and r_{i_z-1} , respectively; the first children of N_P and N_R , respectively, is labeled

$$A_1 \rightarrow \gamma_1 \text{ and } A_2 \rightarrow \gamma_2,$$

such that these two productions are identical; thus, $p_{i_z} = r_{i_z}$;

(b) if $\alpha_{P,i_z} \neq \epsilon$, then

$$A_{P,i_z-1} \rightarrow \alpha_{P,i_z-1} \gamma_{P,i_z-1} = A_{P,i_z} \rightarrow \alpha_{P,i_z} \gamma_{P,i_z}$$

such that

$$|\alpha_{P,i_z-1}| + 1 = |\alpha_{P,i_z}|;$$

note that the node corresponding to p_{i_z-1} is identical to the node corresponding to r_{i_z-1} , and that the least sibling of p_{i_z-1} that is greater than p_{i_z-1} corresponds to p_{i_z} , and is labeled by the grammar symbol Y , such that

$$\alpha_{P,i_z-1} Y = \alpha_{P,i_z}; \quad (27)$$

the node corresponding to r_{i_z-1} must have a sibling identical to the one corresponding to p_{i_z} ; this sibling corresponds to r_{i_z} , so we conclude from (27) that $p_{i_z} = r_{i_z}$.

In all of these cases, we see that we have $P_k = R_k$.

We have established a result that allows us to claim that $U_1 = U_2$ and that $J_1 = J_2$.

Assume that $D_1 \neq D_2$; we clearly have that $Q_D(D_1) = Q_D(D_2)$ —from this, we will derive a contradiction. Let H be the height of the last element of $Q_D(D_1)$; thus

$$|D_1| = H = |D_2|.$$

Let $D_1 = (L_1, L_2, \dots, L_H)$ and let $D_2 = (L'_1, L'_2, \dots, L'_H)$, and let k_d be the least index such that $L_{k_d} \neq L'_{k_d}$. There are two cases.

1. Assume that $k_d = 1$. Let $Q_D(D_1) = (Y_{L,1}, Y_{L,2}, \dots, Y_{L,n_Y})$; we have that

$$\gamma_L = \mathcal{L}(\mathcal{R}(Y_{L,1})) \mathcal{L}(\mathcal{R}(Y_{L,2})) \dots \mathcal{L}(\mathcal{R}(Y_{L,n_Y})).$$

Since $L_1 = (\gamma_L, |Q_D(D_1)|)$ and let $L'_1 = (\gamma_L, |Q_D(D_2)|)$, and since $|Q_D(D_1)| = |Q_D(D_2)|$, we conclude that $L_1 = L'_1$.

2. Assume that $k_d > 1$. Let T_d and T'_d be the final elements of $Q_D(D_1)$ and $Q_D(D_2)$, respectively. In this case, let N_d be the node in T_d such that N_d is the greatest node with exactly $|D_1| - k_d$ ancestors; let N'_d be similarly defined for T'_d . The parents of the nodes N_d and N'_d , corresponding as they do to the identical links L_{k_d-1} and L'_{k_d-1} , are identical, and identically labeled with the

production $A_L \rightarrow \alpha_L$. Both L_{k_d} and L'_{k_d} are links, either partial-downward or terminal, for α_L . Let the number of siblings of N_d be n_d , which is also the number of siblings of N'_d . Clearly,

$$\begin{aligned} L_{k_d} &= (\alpha_L, n_d), \text{ and} \\ L'_{k_d} &= (\alpha_L, n_d). \end{aligned}$$

Thus, conclude that $L_{k_d} = L'_{k_d}$.

As we have a contradiction either way, we have therefore that $D_1 = D_2$.

The only other possibility is that $s_1 \neq s_2$. Let P_s be the attach-point of $Q_A(J_1)$, and let $\mathcal{L}(P_s) = A_s \rightarrow \alpha_s X_s \gamma_s$ such that P_s has $|\alpha_s|$ children. The child-string of P_s is clearly α_s , and L_1 (the first element of D_1 or D_2) is a link for γ_s . Therefore,

$$s_1 = [A_s \rightarrow \alpha_s X_s \cdot \gamma_s] = s_2.$$

Since $(U_1, J_1, s_1, D_1) = (U_2, J_2, s_2, D_2)$, we have that R is injective, and therefore, a bijection. \blacksquare

Theorem 10. *Let $G = (\Sigma, N, P, S)$ be an LR(1) grammar; let βB be a viable prefix followed by a , for $B \in N$, and let \mathbf{P} be a parse-path for βB . If π is a production, then*

$$L_{V,\mathbf{P}}(\pi) = M(R(\mathbf{P}), \pi).$$

Proof. Let $\mathbf{P} = (U, J, s, D)$.

Let $U = (u_1, u_2, \dots, u_n)$, let $J = (j_1, j_2, \dots, j_m)$, let $s = [E \rightarrow \eta \cdot \theta]$, and let $D = (L_1, L_2, \dots, L_p)$. For $1 \leq i_n \leq n$, let $u_{i_n} \equiv [A_{i_n} \rightarrow \alpha_{i_n} \cdot \gamma_{i_n}, a]$; likewise, for $1 \leq i_m \leq m$, let $j_{i_m} \equiv [C_{i_m} \rightarrow \delta_{i_m} \cdot \zeta_{i_m}]$; finally, for $1 \leq i_p \leq p$, we let $L_{i_p} \equiv (\theta_{i_p}, k_{i_p})$. Let T be the proper simplified tree for βB followed by a . Let $T_U = Q_U(U)$, let $T_J = Q_A(J)$, and let $Y = Q_D(D)$.

Let $\pi = A \rightarrow \alpha$ be a production, and let $L_{V,\mathbf{P}}(\pi) = n_V$. We will show that

$$L_{V,\mathbf{P}}(\pi) \leq M(T, \pi). \quad (28)$$

If $n_V = -1$, then $M(T, \pi) \geq n_V$ very trivially. So assume that $n_V \neq -1$. There are six possibilities.

1. Assume that $V_{II}(\pi) = n_V$. In this case, there is some i_u such that

$$A_{i_u} \rightarrow \alpha_{i_u} \gamma_{i_u} = \pi.$$

Now

$$u_{i_u} = [A_{i_u} \rightarrow \alpha_{i_u} \cdot \gamma_{i_u}, a];$$

let $l_u = i_u - |\alpha_{i_u}|$; evidently,

$$u_{l_u} = [A_{l_u} \rightarrow \cdot \gamma_{l_u}, a];$$

which is to say that $\alpha_{l_u} = \epsilon$. When we evaluate Q_U on the item u_{l_u} , a node labeled π gets added to $Q_U((u_1, u_2, \dots, u_{l_u-1}))$; let this node be N_u . Since N_u is an ancestor of B , but not A , then by Definition 45, we have that $M(T, \pi) = |\gamma_{l_u}| + 1$.

2. Assume that $V_{\Pi,\epsilon}(\pi) = n_V$. This means that π is used in the derivation

$$\gamma_{i_\epsilon} \xrightarrow{*} \epsilon,$$

for some $1 \leq i_\epsilon \leq n$. There must, according to Definition 59, be some node labeled π in one of the parse trees attached to T_U to form $T_{U,0}$, as specified in the same Definition. Let $N_{U,0}$ be this node. Since $N_{U,0}$ shares an ancestor (corresponding to the attach-point of T_J) with both B and A , we have that $M(T, \pi) = |\alpha| + 1$.

3. Assume that $V_\Phi(\pi) = n_V$. This means that there is some $1 \leq k_J \leq m$ such that $C_{k_J} \rightarrow \delta_{k_J} \zeta_{k_J} = \pi$. We have in this case that

$$|\delta_{k_J}| + 1 = n_V. \quad (29)$$

As every production node of T_J is the greatest of its siblings, we see that only the greatest child of a production node can be autoancestral to B . Let $k' = k_J - |\delta_{k_J}|$; When we evaluate Q_A on $j_{k'}$, we add a node labeled π to T_J —let this node be N_J . There will be at least $|\delta_{k_J}|$ children of N_J . Thus, by Definition 45,

$$H_{T_J}(N_J) = |\delta_{k_J}| + 1,$$

and so $M(T, \pi) = |\delta_{k_J}| + 1$.

4. Assume that $V_\Psi(\pi) = n_V$. This means that there is some $1 \leq h_V \leq p$ such that

$$P(L_{h_V}, L_{h_V+1}) = \pi,$$

where

$$k_{h_V+1} = n_V. \quad (30)$$

When evaluating Q_D on L_{h_V} , we create a sequence of trees: the final tree T_D in this evaluation will have a root labeled π , such that this root has k_{h_V+1} children; by Definition 45, the function

$$H_{T_D}(\pi) = k_{h_V+1}.$$

We have, by (30), that

$$M(T_D, \pi) = n_V.$$

5. Assume that $V_{\Psi,\epsilon}(\pi) = n_V$. This means that there is some $1 \leq h'_V \leq p$, where, writing $\theta_{h'_V} = X_1 X_2 \dots X_{N_D}$, we have that π is used in the derivation

$$X_{l_D} \xrightarrow{*} \epsilon, \quad (31)$$

for some $1 \leq l_D \leq N_D$. When, according to Definition 56, we evaluate Q_D on the downward link $L_{h'_V}$, we create a parse tree for derivation (31); in this parse tree, there will be a node $P_{D,\epsilon}$ which is labeled π . This node $P_{D,\epsilon}$ shares an ancestor with both B and A ; moreover, $B < P_{D,\epsilon} < A$. Therefore, we have

$$H_T(\pi) = |\alpha| + 1 = n_V.$$

6. Assume that $V_\Omega(\pi) = n_V$. In this case, we first note that $\pi = E \rightarrow \eta\theta$. When we join the two trees T_J and T_U to the trees in Y , we find that the node that was the attach-point of T_U now has $|\eta| - 1$ children inherited from T_J , one child corresponding to either the sidelink or the root of T_U , and one child for

each of the elements of Y . There children are added in the order listed: note that the final element of Y corresponds to a node that is autoancestral A . Let

$$i_s = m - |\eta| - 1;$$

since $|\eta| > 1$, we know that

$$j_{i_s} = [E \rightarrow \cdot \eta \theta].$$

Let P_s be the node corresponding to this item (added during the evaluation of Q_A); we have already established that P_s is an ancestor of A with $|\eta| + |Y|$ children, the last of which is autoancestral to A . Therefore, according to Case 6 of Definition 45, we have that

$$H_T(P_s) = n_V.$$

Therefore, we have established (28).

Let $\phi = C \rightarrow \gamma$ be some production, and let $M(T, \phi) = n_T$. We dispense with the possibility that $n_T = -1$, as it is trivial; thus, assume that $n_T > -1$. There must be some node P_ϕ in T such that $\mathcal{L}(P_\phi) = \phi$, such that,

$$H_T(P_\phi) = n_T.$$

There are three cases to consider for the relationship of P_ϕ with the nodes B and A (with some subcases).

1. Assume that P_ϕ is an ancestor of B , while P_ϕ is not an ancestor of A . By the construction of $T_{U,0}$ in Definition 39, we can see that P_ϕ has $|\gamma|$ children and so, the value of $H_T(P_\phi)$ is $|\gamma| + 1$. There must be some item u_B in U corresponding to P_ϕ ; this item will be of the form

$$u_B = [C \rightarrow \cdot \gamma, a].$$

By Definition 38, Condition 1, we can see that $V_{\Pi}(\phi) = |\gamma| + 1$.

2. Assume that P_ϕ is an ancestor of A . In this case, we have that P_ϕ corresponds to some element of (U, J, s, D) , according to one of the following possibilities.
 - (a) There could be some node in one of the elements of Y corresponding to P_ϕ . This node, which we call P_Y , is in the last element of Y , a tree which we label T_Y . Let the number of ancestors of P_Y in T_Y be h_a . When evaluating Q_D on D , we add the node P'_Y —corresponding to P_Y —during the evaluation of Q_D on L_{h_a+1} . This node P'_T will be the root of a subtree of T_Y ; a subtree which was created when evaluating Q_D on L_{h_a+1} . Since P'_T has k_{h_a+1} children,

$$n_T = k_{h_a};$$

thus, since $P(L_{h_a+1}, L_{h_a+2}) = \phi$, we have therefore that

$$V_{\Psi}(D) = k_{h_a+1} = n_T.$$

(b) Assume that the child of P_ϕ which is autoancestral of A does not correspond to any node in T_J . Let P_A be the node in T_J corresponding to P_ϕ ; note that P_A has $n_T - 1$ children in T_J . We added P_A to T_J when evaluating Q_A on the item j_{m-n_T+1} . Let j_{m-n_T+1} be of the form

$$[D_J \rightarrow \cdot \zeta_J \eta_J];$$

evidently, j_m is of the form

$$[D_J \rightarrow \zeta_J \cdot \eta_J].$$

We know that $|\eta_J| > 1$, and so we write $\eta_J \equiv X_J \eta_S$; we must have that

$$s = [D_J \rightarrow \zeta_J X_J \cdot \eta_S] = [E \rightarrow \eta \cdot \theta].$$

Now, P_ϕ will have one child corresponding to each of the children of P_A ; additionally, it will have one P_c , such that P_c is either labeled B , or P_c is the node corresponding to the root of $Q_U(U)$; finally, it will have one child for each of the elements of Y . Since $|Y| = k_1$, we have that

$$n_T = |\eta| + k_1;$$

thus, by Definition 38, we conclude that

$$V_\Omega(\phi) = n_T.$$

(c) Assume that we do not have Case 2b, and that there is some item in J corresponding to P_ϕ . This item is of the form

$$j_{i_A} = [C \rightarrow \cdot \gamma].$$

As P_ϕ has $n_T - 1$ children, we must have that

$$j_{i_A+n_T-1} = [C \rightarrow \delta_J \cdot \zeta_J],$$

where

$$|\delta_J| = n_T - 1$$

such that $\delta_J \zeta_J = \gamma$. Therefore, according to Definition 38,

$$V_\Phi(\phi) = |\delta_J| + 1 = n_T.$$

In all three of these cases, we find that

$$L_V(\phi) \geq M(T, \phi).$$

3. Finally, assume that P_ϕ shares an ancestor with both B and A , but is an ancestor of neither, yet $B < P < A$. There are two ways that this can happen.

(a) Assume that P_ϕ shares an ancestor with B but not A , such that $B < P < A$. Among all such ancestors, there is one, which we label Q_B , such that Q_B is the root of a subtree that shares no nodes with $Q_U(U)$, save Let $T_{U,0}$ be an in Definition 59. Let the node in $T_{U,0}$ corresponding to P_ϕ be $P_{U,0}$. Let R_B be that child of Q_B that is autoancestral to B ; by the construction of $T_{U,0}$, we can see that the subtree rooted at R_B corresponds to one of the derivations

$$\gamma_{k_0} \xrightarrow{*} \epsilon$$

for some $1 \leq k_0 \leq n$. Therefore,

$$V_{\Pi, \epsilon}(\phi) = |\gamma| + 1.$$

(b) Assume that P_ϕ shares no ancestor with B that is not also an ancestor of A . Let $Y = (V_1, V_2, \dots, V_{k_1})$. Let P_ϕ correspond to a node $P_{Y,\epsilon}$ in any of the trees in Y . If $P_{Y,\epsilon}$ is in any of $V_1, V_2, \dots, V_{k_1-1}$, say, V_{h_Y} , then this node $P_{Y,\epsilon}$ is the root of a subtree with yield ϵ in the tree V_{h_Y} . If P_ϕ corresponds to a node in V_{k_1} , since the yield of V_{k_1} is a , and $P_{Y,\epsilon}$ is not an ancestor of A , we have that $P_{Y,\epsilon}$ is again the root of a subtree with yield ϵ . Therefore,

$$V_{Y,\epsilon}(\phi) = |\gamma| + 1.$$

In all three of these cases, we have established that

$$L_{V,P}(\phi) \geq M(T, \phi). \quad (32)$$

Therefore, by (28) and (32), we have that

$$L_{V,P}(\phi) = M(T, \phi). \quad \blacksquare$$

Thus, we may say that P and T_P are “isomorphic” under R , with respect to conservation of productions.

5 Related Work

Going back to the first days of high-level computer languages, the general idea of a computer language whose parser could modify itself—a construction called an “extensible language”—was considered and tried numerous times. However, these efforts were not always met with success. Perhaps it was because compiler construction as a discipline itself was not well understood, or that the appropriate formal language theory had not been developed, or that the extensible compilers were not powerful enough: for whatever reason, extensible languages have largely fallen by the wayside.

One of the first serious extensible language projects was a variant of Algol 60 called IMP [18]. Along with IMP, another well regarded extensible language was ECL [34]. These languages allowed programs to (in modern parlance) specify new productions for the language, and supply a replacement template, much in the manner of a macro definition. The parsers for such languages were apparently complex, ad hoc, and arcane affairs; a programmer wishing to extend such a beast needed to understand a fair bit of the internals of the parser to extend it and understand the cause of problems.

An often expressed goal for an extensible language would be to allow a program to supply a new data type, along with associated infix operations, so that programs dealing with matrices or complex numbers could use the natural syntax. The modern approach to this problem is to use operator overloading in an object-oriented language. This begs the question: why would a programmer want to engage in the difficult endeavour of modifying the parser when a mechanism like operator overloading suffices?

The high point of interest in extensible languages was likely the International Symposium on Extensible Languages. The Proceedings of this Symposium [30] contain several reports on real-world extensible languages: there are many reports on languages like ECL (e.g. [34], [6], and [28]); some more general works on macro systems (e.g. [27], and [15]); and some survey works (e.g. [14], and [12]). The mood was upbeat, but a little over-optimistic; indeed, as Cheatham put it,

extensible languages had delivered: “there exist languages and host systems which fulfill the goals of extensibility” [12]. But not everyone was upbeat: there were reports of failures—not of implementation, but of extensibility itself [32].

The extensible language concept did not disappear, even after the appearance of languages like C++. One example is [9], a language that turned out to be only partially successful, complex to use, and crippled by performance problems.

Some of these systems use a self-modifying compiler, and operate in a single pass, while others use a two-pass compiler. Almost all of them do a textual substitution, at least on the conceptual level. We would therefore consider the study of these systems to be a study of macro systems: if a macro system admits patterns that are more complex than a function call (e.g. the CPP macro system for C and C++ requires macros to be of the form `MACRO_NAME(PARAM1, PARAM2, ...)`), then we may say that the macro system is a **syntactic macro** system—otherwise, we say that the macro system is a **simple macro** system. The study of syntactic macros has continued in its own right, and syntax macros are present in some modern languages, including Scheme [25].

An interesting summary of the issues related to advanced macro systems is due to Brabrand and Schwartzbach [5], who summarize prominent macro systems, and present a new one of their own. The macro system presented in [5] operates on partial parse-trees, which illustrates the necessity of a macro-aware parser.

The aforementioned syntactic macro systems can use macros in very powerful ways, achieving many of the goals of the early extensible languages. It is certainly possible to implement a language with syntax macros using a parser for what we have in the present work termed a transformative parser, which would allow for an implicit macro call. This has been done in [10], where a syntax macro programming language implemented using a transformative LL parser is described. The latter system is powerful enough to extend a functional language into an imperative language (like C), and it avoids the problems associated with many macro systems.

Most likely due to their syntactic simplicity (even austerity), syntax-macros are usually reserved for functional programming languages. That is not to say that they cannot be used for a syntactically-rich language like C. Exactly this was done for C by Weise and Crew [35]; one point of note is that, rather than supply a static template with which to replace the macro invocation, their system allows for the replacement to be generated by running procedural code on the macro parameters; the replacement will be an abstract syntax tree. Allowing the macro body to include code, which will be run (most likely by an embedded interpreter) during compilation, could be called **compile-time computation**. Other macro systems allow this—Scheme most notably. Another system which allows for compile-time computation is C++: it has been discovered that the C++ template system is Turing complete [33].

The choice to allow compile-time computation in the macro body has significant advantages—see [20] for a survey of partial evaluation—but there are many drawbacks. The biggest drawback is the increased complexity of having two languages side by side: the run-time language and the compile-time language.

The present work is part of an effort to make a real-world programming language using a transformative LR(1) parser. This programming language would, it is hoped, prove useful to developers of domain-specific embedded languages (DSEL). The subject of DSEL is of much interest today: for example, see the recent survey piece by Mernik, Heering, and Sloane [24]. In order to develop DSELs,

it is usually necessary to make modifications to the compiler’s source code, a task hopefully made easier by using a programming language with a transformative LR(1) parser at its core.

Another task which requires the modification of a compiler’s source code is the extension of a programming language to add new features: for example, adding aspect-oriented capabilities to Java [3]. This is done often enough (especially with Java in recent years) that software systems dedicated to this task have appeared [26]. Indeed, this was one of the original motivations for early extensible languages [29]. However, for the Java systems mentioned above, the compiler is extended prior to compilation, so we may term this an **offline grammar transformation**. Some other modern systems do allow for transformation during parsing, but a special macro invocation must be used to tell the parser to launch a subparser—i.e. the parse is not self-modifying. This can be done with quoting, as in [22].

The formal basis for the TLR parsing algorithm is the theory of LR(k) languages. Introduced by Knuth [21], his original paper is insightful; another reference for the theory of LR(k) parsing is [2]. Coming at LR(k) parsing from the more practical side is the classic “Dragon Book” [1] (for $k = 1$); this last work is particularly recommended. We base our transformative language on the LR(1) languages because the parsing algorithm for this class of languages is well-suited for a transformative language parser: since a substring of a sentence can define the syntax for some substring immediately to its right, we evidently want to scan sentences from left to right; a backtracking algorithm is undesirable because it is complex and expensive to backtrack past a point at which a grammar transformation occurred; finally, bounded lookahead is important because until a decision is made as to whether or not a grammar transformation will take place at a certain point, it is unknown which (context-free) grammar has the lookahead under its purview. Also, in practice, LR(1) parsers are designed to execute code fragments after reduction by certain productions: this is a natural place to insert the grammar transformation algorithm—as indeed, we have done in the present work.

It is not surprising that a parser for a transformative language based the LR(1) languages has been presented before. Burshteyn [7] formalizes an idea of **modifiable grammars**—roughly equivalent to a transformative grammar. Much of the present work is concerned with allowable transformations: in Section 2.4.1, we saw the negative consequences of admitting completely arbitrary transformations. In [7], the language generated by a modifiable grammar is equivalent to the naive language considered in Section 2.4.1, so it does not avoid those pitfalls.

We do note that, if we only add or remove a few productions to or from a grammar, the LR(1) parsing tables for that grammar do not change “too much.” The algorithm we present in the present work requires the parsing table to be completely regenerated upon acceptance of a grammar transformation. This need not be the case: indeed, in [7], the parser is modified only inasmuch as the grammar transformation (our terminology) requires it; the method of incremental LR(1) parser generation is originally due to Heering, Klint, and Rekers [16].

The canonical LR(1) parser, creating by way of the method of Knuth [21], is often eschewed for the LALR(1) parser, owing to the fact that the LALR(1) parser (if it even exists) has much smaller parsing tables—however, Spector [31] showed how to construct a different LR(1) parser that is often similar in size to the LALR(1) parser. It would probably be a profitable exercise to investigate the use of these techniques in the TLR parsing algorithm for a real-world system.

The conventional view is that programming languages are not context-free:

that is, a program which uses an undeclared identifier is considered to be syntactically well-formed (and hence in the context-free language generated by the grammar) but semantically meaningless.¹ In order to catch this semantic gaffe, the parser performs a separate (at least in principle) semantic analysis phase, which is usually performed by procedural code; an alternative, declarative approach to syntactic and semantic analysis is to instruct the parser to add a new production for an identifier when it is declared, later to be removed when that identifier goes out of scope. A parser which modifies its grammar to remove the need to perform semantic analysis will be herein referred to as an **adaptable grammar**. Two notable attempts at adaptable grammar systems are [8] and [4]; the field is surveyed in [13]. There has been renewed interest in adaptable grammars: see [11] and [19].

We have not been terribly clear about the differences between syntax and semantics: if we (rightly) assume that syntax is what the parser does, then processing of identifiers and scopes is evidently syntax, assuming a powerful enough parser. It is in fact very difficult to delineate syntax and semantics [23].

It should be possible to make many types of systems on top of a TLR parser. In this section, we have discussed: extensible languages, syntax macros, and adaptable grammars. The techniques in the present work should be general enough to be used to achieve any of these three techniques. Systems with these as their goals have not fared well: it is to be hoped that the problem in the past was a lack of understanding of the fundamental parsing issues; which will hopefully be obviated by TLR techniques.

Conclusion

There has been much interest over the years in languages with features—like syntax macros, extensibility, and adaptable grammars—that are incompatible with a parser generated from a static grammar. Numerous ad hoc efforts have been made to make systems with these features and a parser whose grammar is not fixed, with little success. One problem with these earlier attempts is a lack of understanding of the consequences of changing the grammar—a deficiency that this work hopes to address.

Another explanation may be that the features are not audacious enough: why would someone want to trouble themselves with the arcana of the parser to achieve something adequately accomplished by operator overloading, templates, or semantic analysis? If, however, the features are compelling enough, then programmers might be willing to write grammar transformations. We attempted, in the present work, to take some of the mystery out of parsing a transformative language.

Naive transformative languages, while straightforward and occasionally the subject of study, do not allow us to make guarantees about halting. Requiring valid transformations does allow us to make guarantees about halting, although the test for validity is complex. We observe that the utility of this test does not end with ensuring halting of compilation: should Algorithm 2 fail, then we know that the transformation is invalid—a report on which productions were not conserved could be a useful diagnostic; also, the stack represents the different ways that parser is trying to match the sentence, and requiring valid transformations means that, once

¹Although there is the view that a program containing an “undeclared identifier” error is semantically well-formed; we could think of the error message produced by compiling it as the semantic value of the program [21].

a parser starts trying to match a production, that it cannot go back and reinterpret what it already saw.

Many questions remain to be answered.

The central result of the present work is the correctness of Algorithm 1—see Theorem 4—this correctness relies on the transformations emitted by the Δ -machine all being in \mathcal{V} ; is there a larger set of transformations which could fill the role played by \mathcal{V} ?

We allow a full Turing machine to form the basis of a Δ -machine. Theorem 3 of [7] states (in part) that: “Each automatic BUMG (bottom-up modifiable grammar) accepts a context-free language;” in that work, an automatic BUMG is the counterpart of a TLR grammar whose Δ -machine is essentially a finite automaton. We therefore ask: what class of languages are generated by TLR grammars whose Δ -machines are (essentially) finite automata? What class of languages are generated by TLR grammars whose Δ -machines have bounded tapes?

The most important question to answer is this: can a practical transformative programming language be constructed?

References

- [1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers, Principles, Techniques and Tools*. Addison-Wesley, Reading, MA, 1988.
- [2] Alfred V. Aho and Jeffrey D. Ullman. *The Theory of Parsing, Translation, and Compiling*, volume 1 of *Prentice-Hall series in automatic computation*. Prentice-Hall, Englewood Cliffs, NJ, 1972.
- [3] Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Jennifer Lhoták, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. abc: an extensible AspectJ compiler. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 87–98, New York, NY, USA, 2005. ACM Press.
- [4] Pierre Boullier. Dynamic grammars and semantic analysis. Technical Report 2322, INRIA, August 1994.
- [5] Claus Brabrand and Michael I. Schwartzbach. Growing languages with meta-morphic syntax macros. In *PEPM '02: Proceedings of the 2002 ACM SIGPLAN workshop on Partial evaluation and semantics-based program manipulation*, pages 31–40, New York, NY, USA, 2002. ACM Press.
- [6] Benjamin M. Brosgol. An implementation of ECL data types. In *Proceedings of the international symposium on Extensible languages*, pages 87–95, New York, NY, USA, 1971. ACM Press.
- [7] Boris Burshteyn. Generation and recognition of formal languages by modifiable grammars. *ACM SIGPLAN Notices*, 25(12):45–53, December 1990.
- [8] Boris Burshteyn. On the modification of the formal grammar at parse time. *SIGPLAN Not.*, 25(5):117–123, 1990.
- [9] S. Cabasino, P. S. Paolucci, and G. M. Todesco. Dynamic parsers and evolving grammars. *SIGPLAN Not.*, 27(11):39–48, 1992.
- [10] Luca Cardelli, Florian Matthes, and M. Abadi. Extensible syntax with lexical scoping. Technical Report 121, Digital Systems Research Center, 1994.

- [11] Adam Carmi. Adapser: An LALR(1) adaptive parser. The Israeli Workshop on Programming Languages and Development Environments, 2002.
- [12] T. E. Cheatham, Jr. Extensible language - where are we going. In *Proceedings of the international symposium on Extensible languages*, pages 146–147, New York, NY, USA, 1971. ACM Press.
- [13] H. Christiansen. A survey of adaptable grammars. *SIGPLAN Not.*, 25(11):35–44, 1990.
- [14] J. J. Duby. Extensible languages: A potential user’s point of view. In *Proceedings of the international symposium on Extensible languages*, pages 137–140, New York, NY, USA, 1971. ACM Press.
- [15] Michael Hammer. An alternative approach to macro processing. In *Proceedings of the international symposium on Extensible languages*, pages 58–64, New York, NY, USA, 1971. ACM Press.
- [16] J. Heering, P. Klint, and J. Rekers. Incremental generation of parsers. *SIGPLAN Not.*, 24(7):179–191, 1989.
- [17] R. N. Horspool. Incremental generation of LR parsers. *Comput. Lang.*, 15(4):205–223, 1990.
- [18] Edgar T. Irons. Experience with an extensible language. *Commun. ACM*, 13(1):31–40, 1970.
- [19] Quinn Tyler Jackson. Some theoretical and practical results in context-sensitive and adaptive parsing. *Progress in Complexity, Information, and Design*, 1(4), December 2002.
- [20] Neil D. Jones. An introduction to partial evaluation. *ACM Comput. Surv.*, 28(3):480–503, 1996.
- [21] Donald E. Knuth. On the translation of languages from left to right. In *Selected Papers on Computer Languages*, chapter 15, pages 327–360. Center for the Study of Language and Information, Stanford, CA, 2003.
- [22] Michel Mauny and Daniel de Rauglaudre. Parsers in ML. In *LFP ’92: Proceedings of the 1992 ACM conference on LISP and functional programming*, pages 76–85, New York, NY, USA, 1992. ACM Press.
- [23] Brian Meek. The static semantics file. *SIGPLAN Not.*, 25(4):33–42, 1990.
- [24] Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4):316–344, 2005.
- [25] IV N. I. Adams, D. H. Bartley, G. Brooks, R. K. Dybvig, D. P. Friedman, R. Halstead, C. Hanson, C. T. Haynes, E. Kohlbecker, D. Oxley, K. M. Pitman, G. J. Rozas, Jr. G. L. Steele, G. J. Sussman, M. Wand, and H. Abelson. Revised⁵ report on the algorithmic language scheme. *SIGPLAN Not.*, 33(9):26–76, 1998.
- [26] Andrew C. Myers Nathaniel Nystrom, Michael R. Clarkson. Polyglot: An extensible compiler framework for Java. In *Proceedings of the 12th international Conference on Compiler Construction*, volume 2622 of *Lecture Notes in Computer Science*, pages 138–152, Heidelberg, January 2003. Springer Berlin.

- [27] M. G. Notley. A model of extensible language systems. In *Proceedings of the international symposium on Extensible languages*, pages 29–38, New York, NY, USA, 1971. ACM Press.
- [28] Charles J. Prenner. The control structure facilities of ECL. In *Proceedings of the international symposium on Extensible languages*, pages 104–112, New York, NY, USA, 1971. ACM Press.
- [29] Jean E. Sammet. Application of extensible languages to specialized application languages. In *Proceedings of the international symposium on Extensible languages*, pages 141–143, New York, NY, USA, 1971. ACM Press.
- [30] Stephen A. Schuman, editor. *Proceedings of the international symposium on Extensible languages*, New York, NY, USA, 1971. ACM Press.
- [31] D. Spector. Efficient full LR(1) parser generation. *SIGPLAN Not.*, 23(12):143–150, 1988.
- [32] Thomas A. Standish. PPL - an extensible language that failed. In *Proceedings of the international symposium on Extensible languages*, pages 144–145, New York, NY, USA, 1971. ACM Press.
- [33] Todd L. Veldhuizen. C++ templates as partial evaluation. In Olivier Danvy, editor, *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 13–18, New York, NY, USA, 1999. ACM Press.
- [34] Ben Wegbreit. An overview of the ECL programming system. In *Proceedings of the international symposium on Extensible languages*, pages 26–28, New York, NY, USA, 1971. ACM Press.
- [35] Daniel Weise and Roger Crew. Programmable syntax macros. In *PLDI '93: Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation*, pages 156–165, New York, NY, USA, 1993. ACM Press.
- [36] Wikipedia. Canonical LR parser—Wikipedia, the free encyclopedia, 2006. [Online; accessed 28-February-2006].